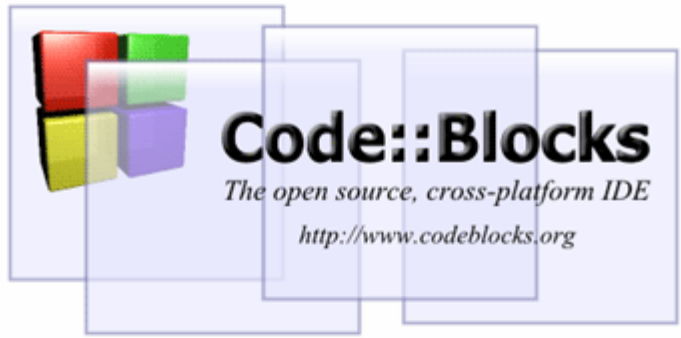


Code::Blocks
for C/C++ Novice

Chipset

Copyright (C) 2008-2010 Chipset
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

目 录

封皮……0
目录……1
前言……2
1. 安装Code::Blocks……3
1.1 下载……3
1.2 安装……3
2. Code::Blocks的编程环境配置……7
2.1 环境……7
2.2 编辑器……12
2.3 编译器和调试器……13
3. 编写程序……17
3.1 创建一个工程……17
3.2 添加和删除文件……20
3.3 编辑文件……25
3.4 编译程序……30
3.5 调试程序……43
3.6 阅读别人编写的程序……74
4. 附录……79
4.1 Linux下安装Code::Blocks……79
4.2 Mac OS X下安装Code::Blocks……80
4.3 Code::Blocks搭配高版本gcc编译器……80
4.4 安装配置boost……82
特别说明……87

前 言

用高级计算机语言，例如C、C++，编写的程序，需要经过编译器编译，才能转化成机器能够执行的二进制代码。然而，把头脑中的思想转变成能够正常工作的计算机程序需要付出一定的努力和时间，因为为了让程序能够达到我们想要的结果，我们往往需要反复修改代码。本书的目的是帮助初学者学习组织程序编码逐步隔离并发现程序中的逻辑错误。通过本书，您可以学会怎么一步步的跟踪代码，找到问题出在什么地方，搞明白为何您的程序不能正常运行，这个过程称谓调试程序。手工跟踪能够有效的帮助初学者找到bug出在什么位置，消除bug，让程序正常运行。自动化的工具同样也能够帮助您跟踪程序，尤其当程序很复杂时效果更加明显，这种工具叫做调试器。调试器能够让运行中的程序根据您的需要暂停，查看程序怎么运作的。有些调试器是以命令行的形式工作的，较新的调试器有些具备好的图形界面，调试器能够方便的帮助您看到您定义的变量状态。基于图形界面的调试器是集成开发环境(IDE，即Integrated Development Environment)的一部分。本书的作用就是帮助您学习使用这种环境以便更好的掌握编程技巧。

一个调试器并不能解决您程序中出现的问题，它仅仅是一种帮助您编程的工具。您首先应该运用您手中的纸和笔分析程序，搞清到底怎么回事，一旦确定错误大致出在什么位置，便可以用调试器观察您的程序中特定变量的值。通过观察这些代码，可以了解到您的程序是怎么一步步执行的。

C/C++的IDE非常多，对于学习C/C++语言的朋友而言，用什么IDE可能并不重要，重要的是学习C/C++语言本身，不过，会用一款自己习惯的IDE进行程序的编写和调试确实很方便。

本书主要论述一款开源、免费、跨平台的集成开发环境Code::Blocks的安装、配置、以及程序的调试和编译等。Code::Blocks支持十几种常见的编译器，安装后占用较少的硬盘空间，个性化特性十分丰富，功能十分强大，而且易学易用。我们这里介绍的Code::Blocks集成了C/C++编辑器、编译器、和调试器于一体，使用它可以很方便的编辑、调试和编译C/C++应用程序。Code::Blocks具有很多实用的个性化特性，这里只会简单介绍少数几个常用的特性。

我们希望本书能够帮助您体验编程的乐趣的同时也能帮助您提高调试和编写程序的基本功。

如欲了解更多有关Code::Blocks的信息，请访问Code::Blocks的官方网站http://www.codeblocks.org。

1. 安装Code::Blocks

1.1 下载

为了安装Code::Blocks IDE，首先需要下载它们。如果您使用的是Windows 2000 或 Windows XP 或 Windows Vista操作系统，从下面地址下载Code::Blocks8.02(目前来说，8.02是最新的版本)这个IDE：

http://downloads.sourceforge.net/codeblocks/codeblocks-8.02mingw-setup.exe

以上地址下载的文件中包含了MinGW它(内嵌了GCC编译器和gdb调试器)

如果您仅仅希望把Code::Blocks当作编辑器使用，或者打算自己配置编译器和调试器的话，可以下载不带MinGW的版本，到下面的地址去下载。

http://downloads.sourceforge.net/codeblocks/codeblocks-8.02-setup.exe

本书的作者建议初学C/C++的朋友下载内置MinGW的版本，这样不致于花费太多时间配置编译器和调试器，从而把大部分时间用于学习调试和编写程序。待将来您熟悉了Code::Blocks，再搭配高版本的MinGW或者其它编译器一起使用。

如果您使用Mac OS X 或Linux操作系统，请参阅附录A中的安装说明。

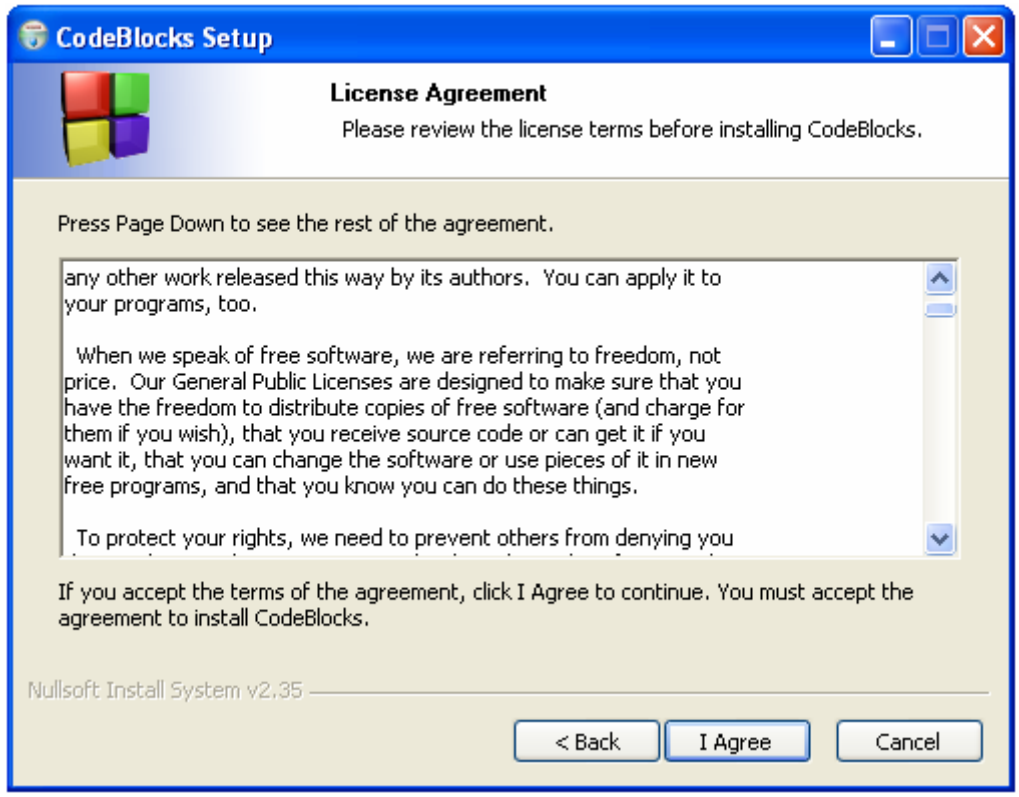
1.2 安装

安装过程可以参照如下步骤进行(以在笔者的英文版Windows XP Professional SP2操作系统上安装Code::Blocks8.02为例)。

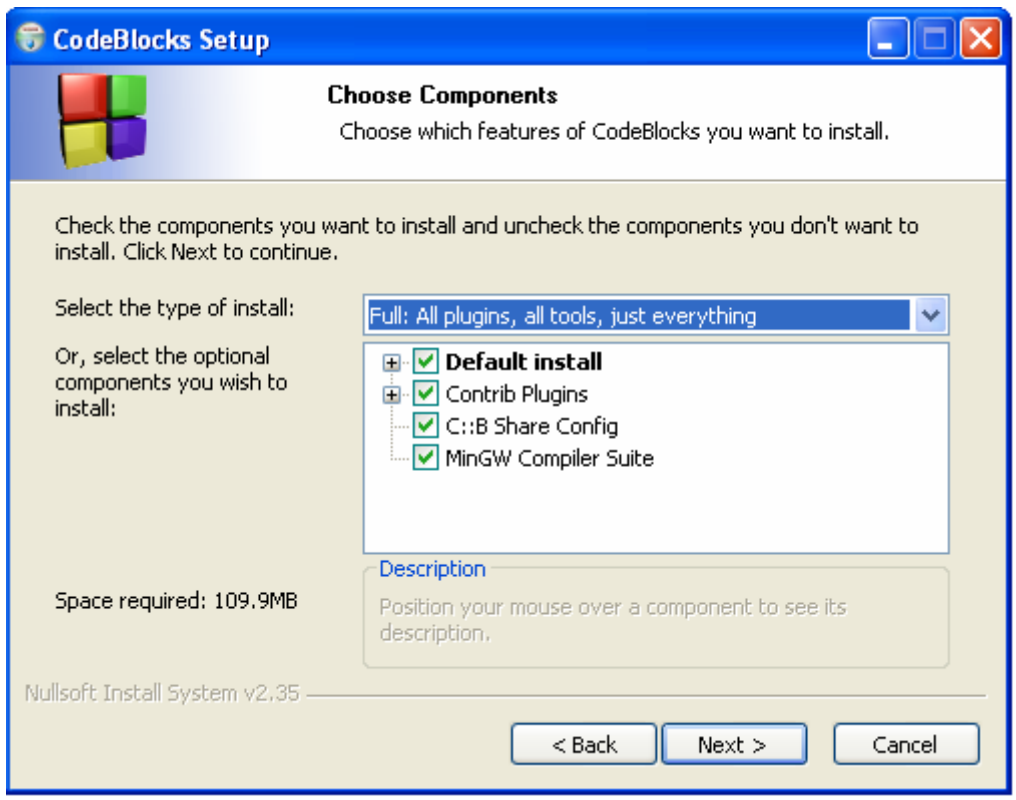
运行下载后的安装文件进入左下图界面。



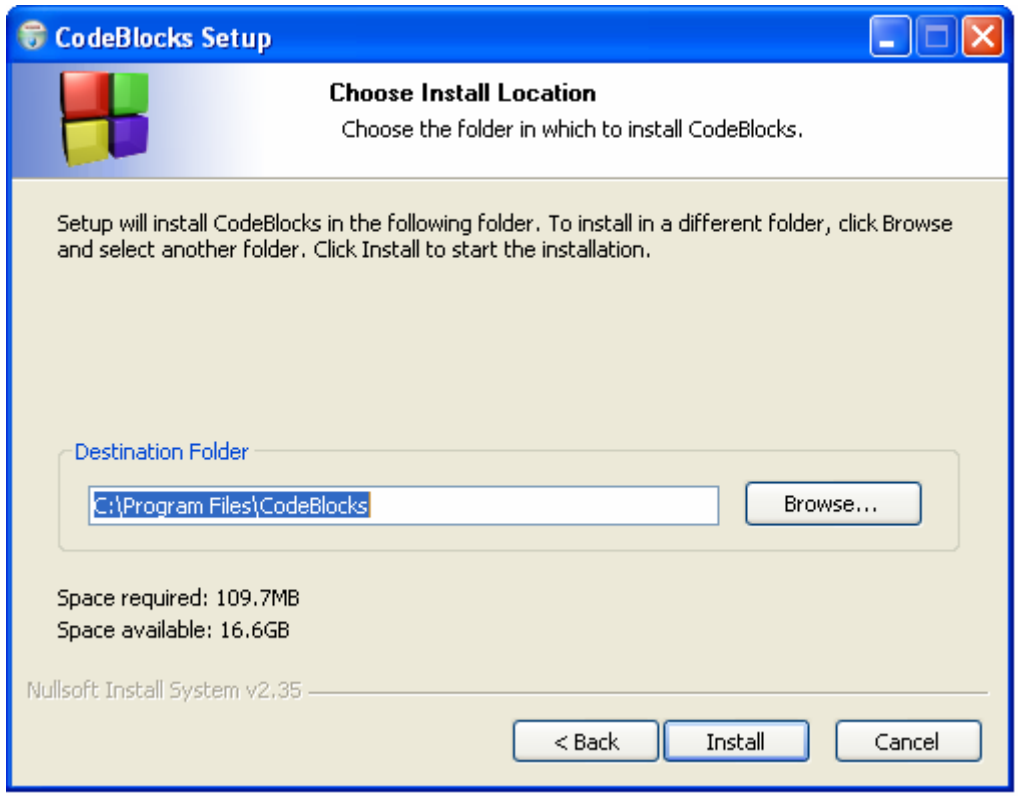
再用鼠标点击Next按钮，可以进入下图界面。



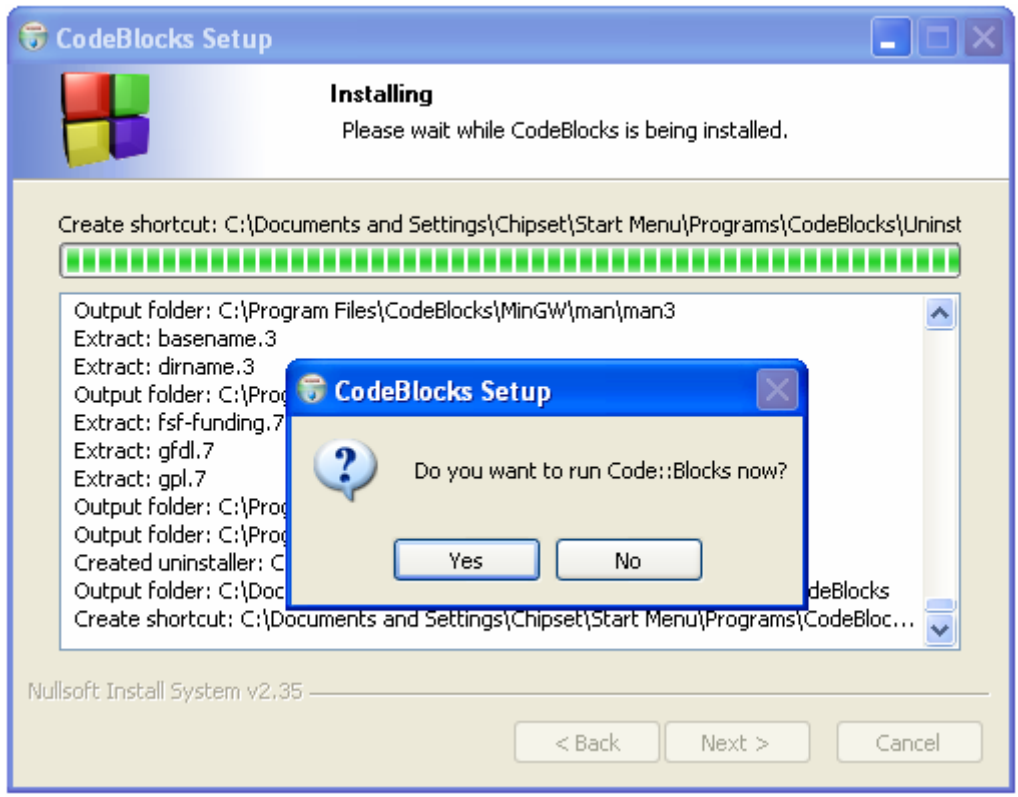
用鼠标选择 I Agree 按钮，进入如下图界面。



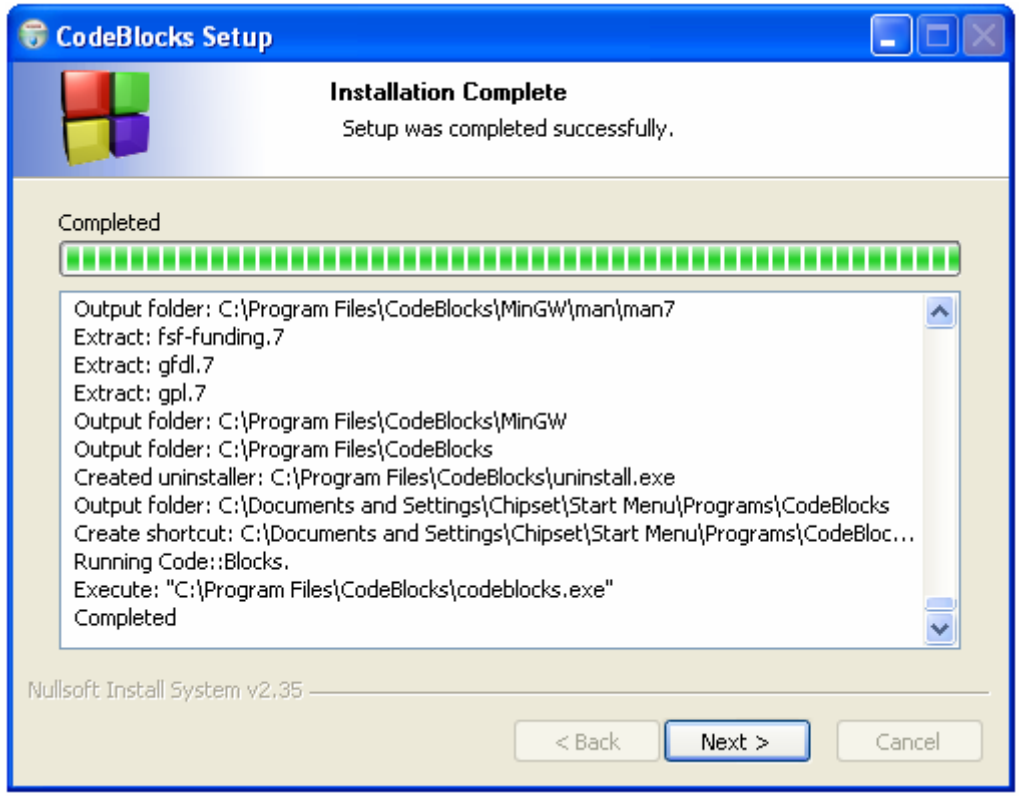
选择全部安装(Full: All plugins, all tools, just everything)，见上图，再点击 Next 按钮，进入一个新界面，如下图。



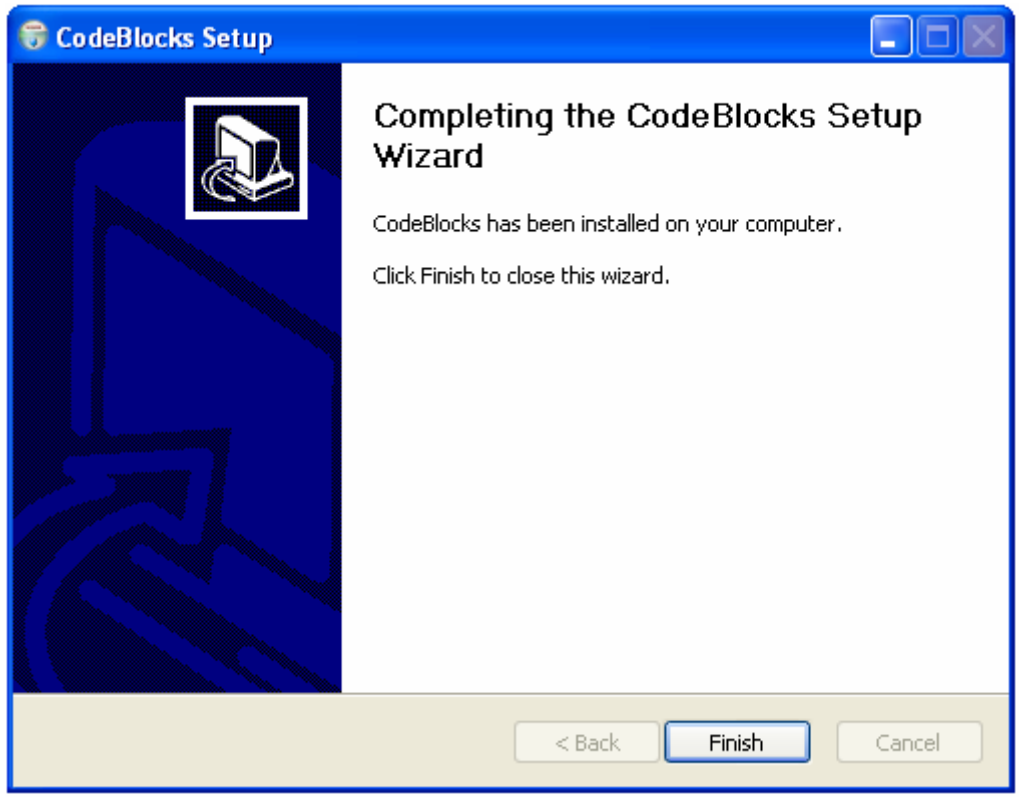
点击 Browse…按钮选好安装路径(默认安装路径为 C:\Program Files\CodeBlocks)，用鼠标选择 Install 按钮，可以看到安装过程正在进行，并弹出一个对话框，见下图。



用鼠标选择 No 按钮，则对话画框关闭，见下图。



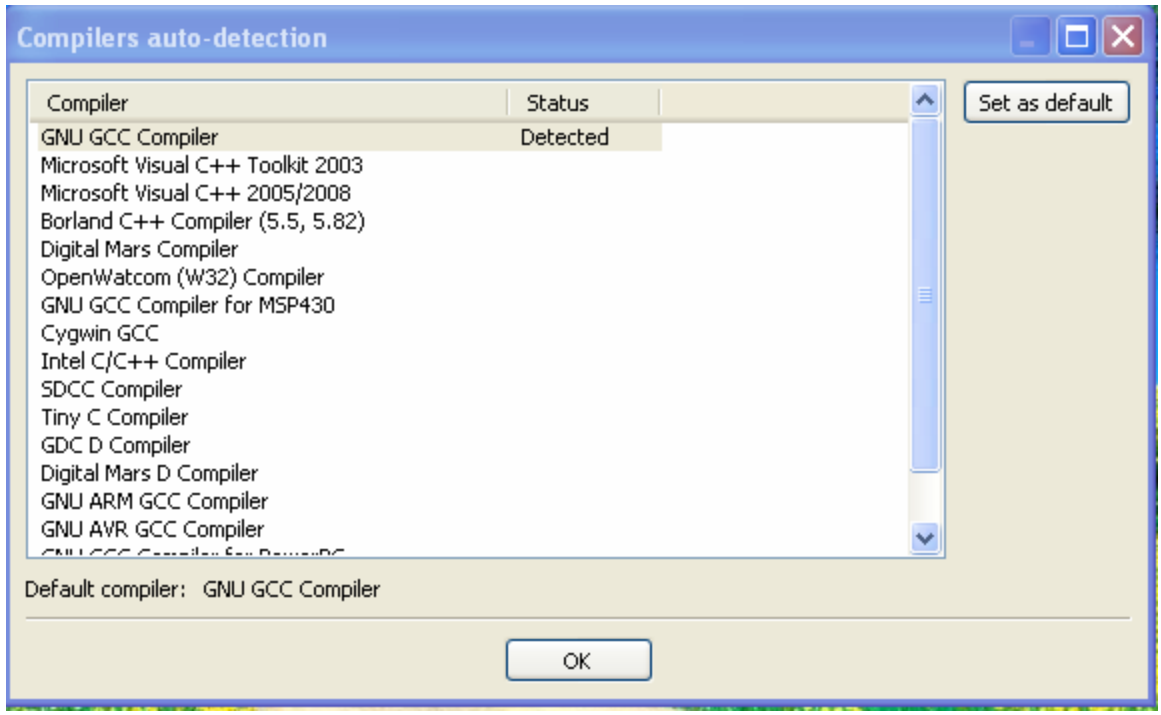
用鼠标选择 Next 按钮，进入界面见下图。



最后用鼠标选择 Finish，则安装过程就完成了。

2. Code::Blocks 编程环境配置

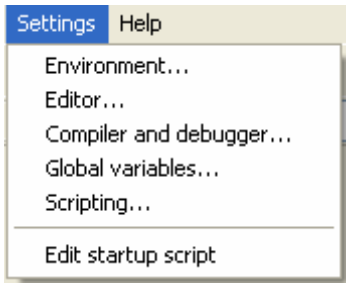
第一次启动 Code::Blocks，可能会出现如下对话框，告诉您自动检测到 GNU GCC Compiler 编译器，用鼠标选择对话框右侧的 Set as default 按钮，然后再选择 OK 按钮，见下图。



假如您的 Code::Blocks 安装正确的话，接下来就进入 Code::Blocks 的主界面，但是会弹出一个标签为 Tips of the Day 的小对话框，见右下图。



把 Show tips at startup 前面的勾去掉，然后选择 Close，这样下次启动就不会再出现这个小对话框。



进入 Code::Blocks 主界面，选择主菜单 Settings，弹出一个窗口，见左上图。然后我们就可以分别对环境(Environment…)，编辑器(Editor…)，编译器和调试器(Compiler and debugger…)三个子菜单进行配置了。

2.1 环境

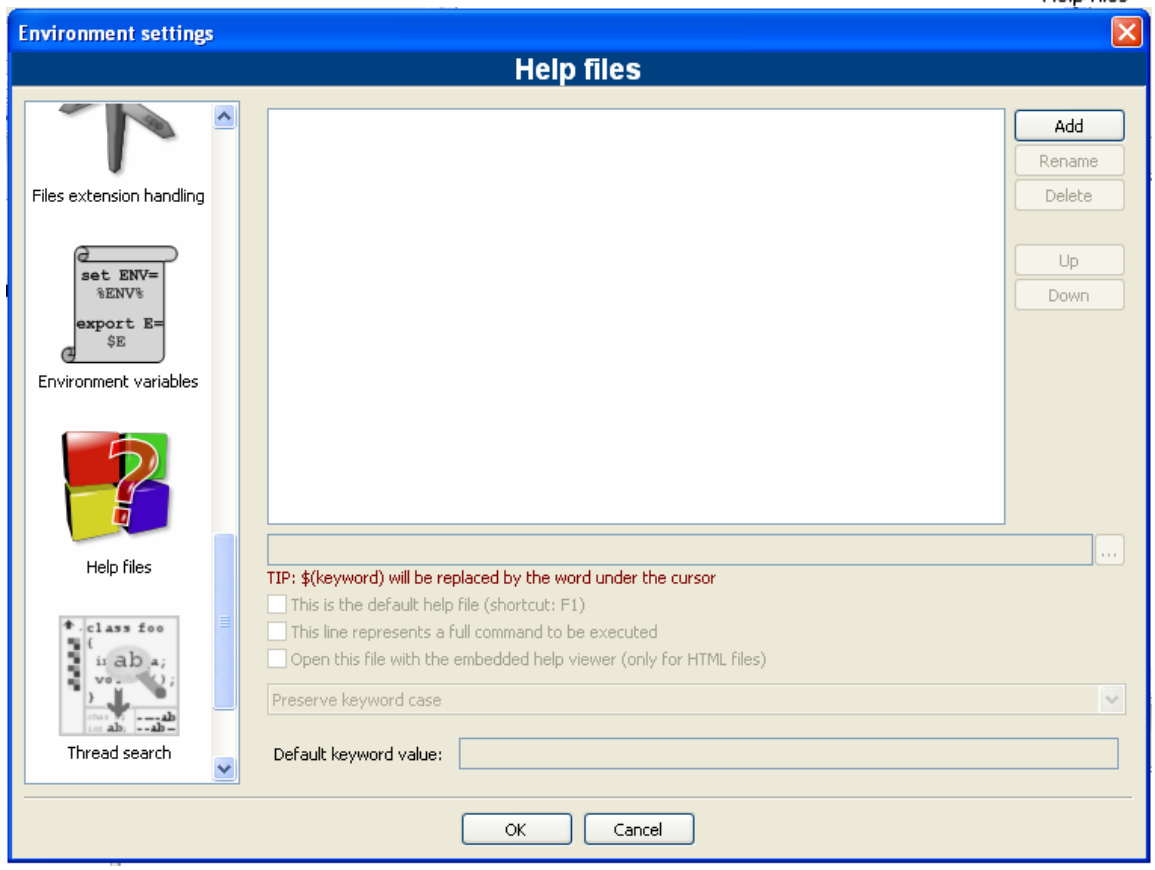
选择主菜单 Settings 下的第一个子菜单 Environment…,会弹出一个窗口，用鼠标拖动左侧的滚动条，可以见到很多带有文字的图标。这些下面带有文字的图标代表了不同的功能按钮。



2.1.1 配置帮助文件

拖动滚动条，用鼠标选择这个图标，见右图。

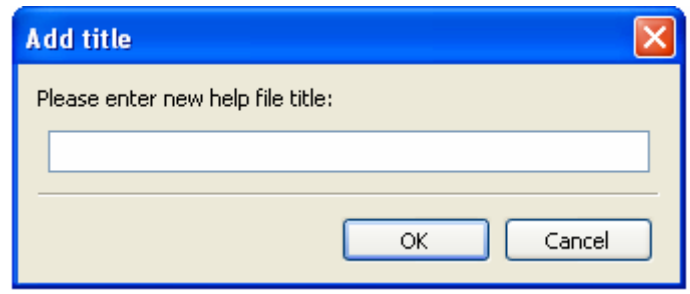
此时会会出现一个对话框界面，见下图。



然后可以添加一些我们可能需要的帮助文件。我们编写基本的C/C++应用程序，仅需要知道C/C++的库函数用法就可以了。如果您没有C/C++语言库函数的文档，请到http://www.cppblog.com/Files/Chipset/cppreference.zip去下载C++ Reference，解压后放到Code::Blocks目录下(也可以放到别处)，以便添加进来编程时方便查阅。可以按照如下步骤进行添加：

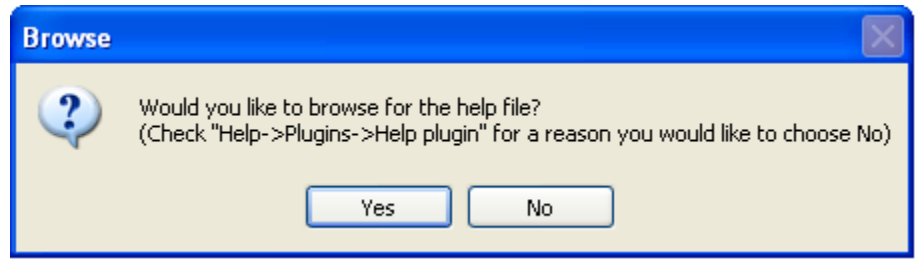
(1) 添加文件

用鼠标点击右上侧的Add按钮，得到对话框见右图。



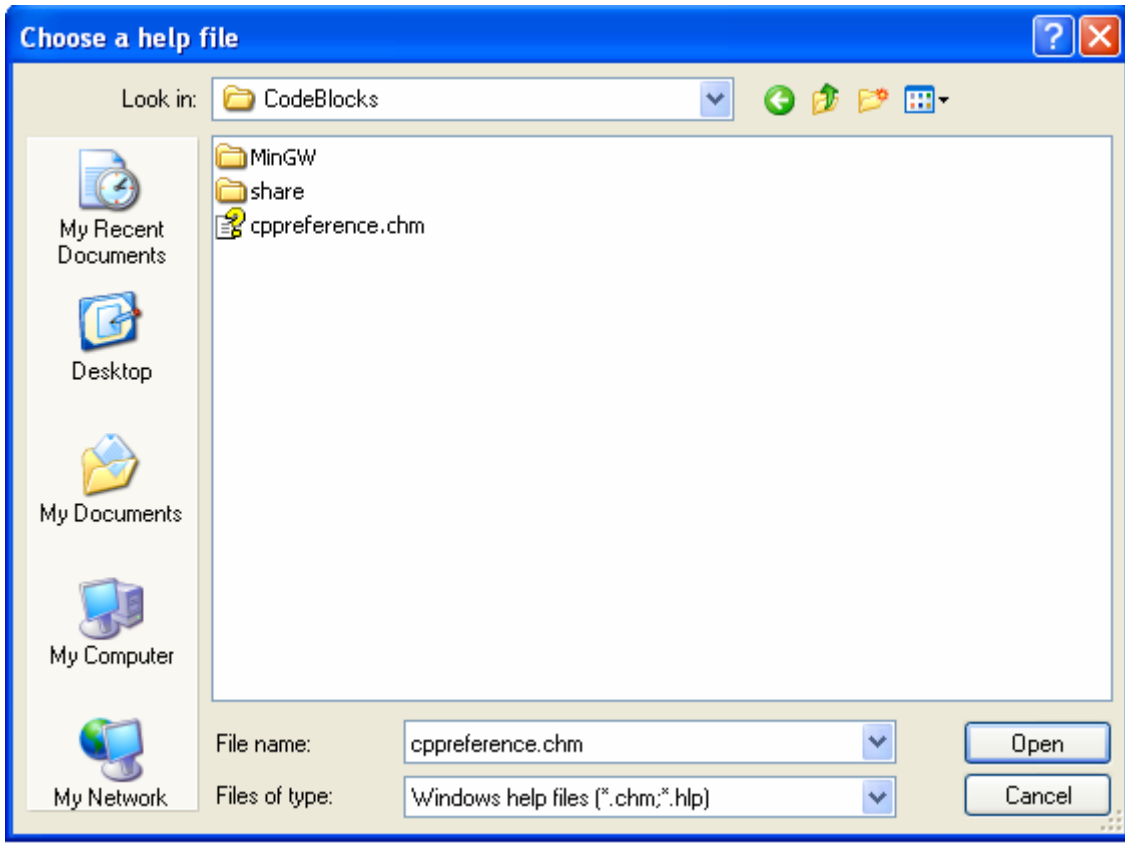
(2) 键入帮助文件题头

给添加的文件取一个题头名，该名字可以跟实际文件名相同，也可以不同，然后选择OK按钮，又弹出一个对话框见



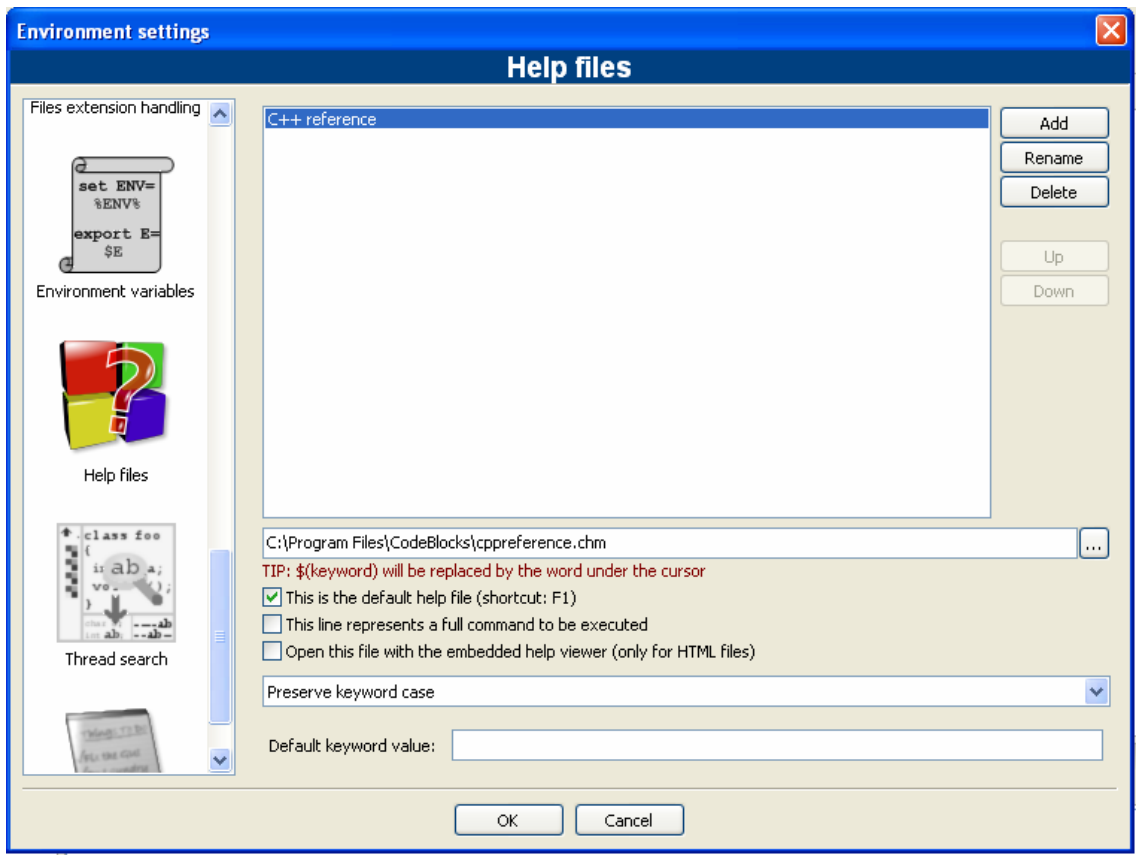
右图。

选择Yes按钮，进入下一步，见下图。



(3) 选择需要打开的文件

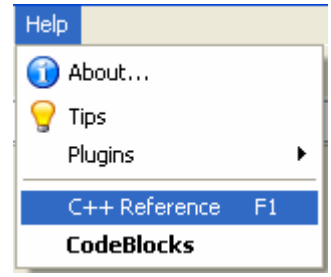
找到帮助文件的路径，选中帮助文件cppreference.chm，然后选择Open就又回到了刚进入Help files的对话框，只不过多了一行字C++ Reference，见下图。



并且有刚加载的文件cppreference.chm的对应路径。可以继续按照上述步骤添加更多帮助文件，也可以用右上侧的按钮Rename对题头C++ Reference进行改名或者用Delete按钮删除此题头。

(4) 使帮助文件可用

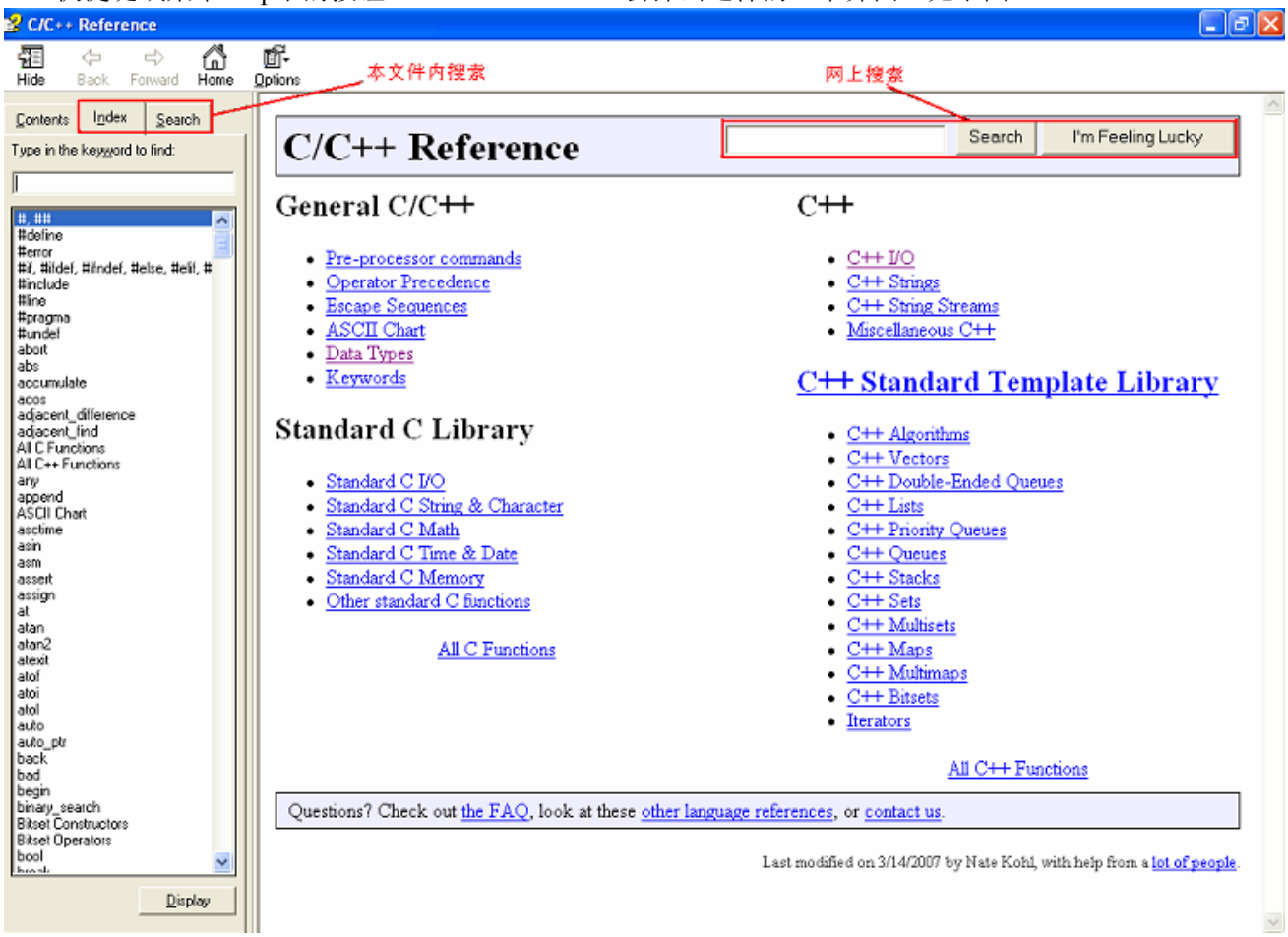
为了方便使用，选中C++ Reference并用鼠标在下面的标签This is the default help file (shortcut: F1)前面的小方框中打勾，见上图，然后再用鼠标点击下面的OK按钮。



(5) 测试帮助文件是否可以成功加载

进入Code::Blocks(如果刚才您未退出Code::Blocks就无需再重新进入)，选择主菜单Help下的C++ Reference F1按钮，如右图。或者按下F1快捷键，就可以成功加载刚才设置需要加载的帮助文件cppreference.chm了。

经过上述这些设置后，Code::Blocks就可以成功加载帮助文件了，按下F1快捷键或菜单Help下的按钮C++ Reference F1会弹出这样的一个界面，见下图。



假如我们需要查阅标准的C++库函数，可以选择左侧的Index或Search按钮键入函数名进行查询。如果您使用的电脑已经联网，在左上部空框内键入要查询的函数名，再用鼠标点击右侧Search按钮就可以进行http://www.cppreference.com网上查询。

2.1.2 自动保存

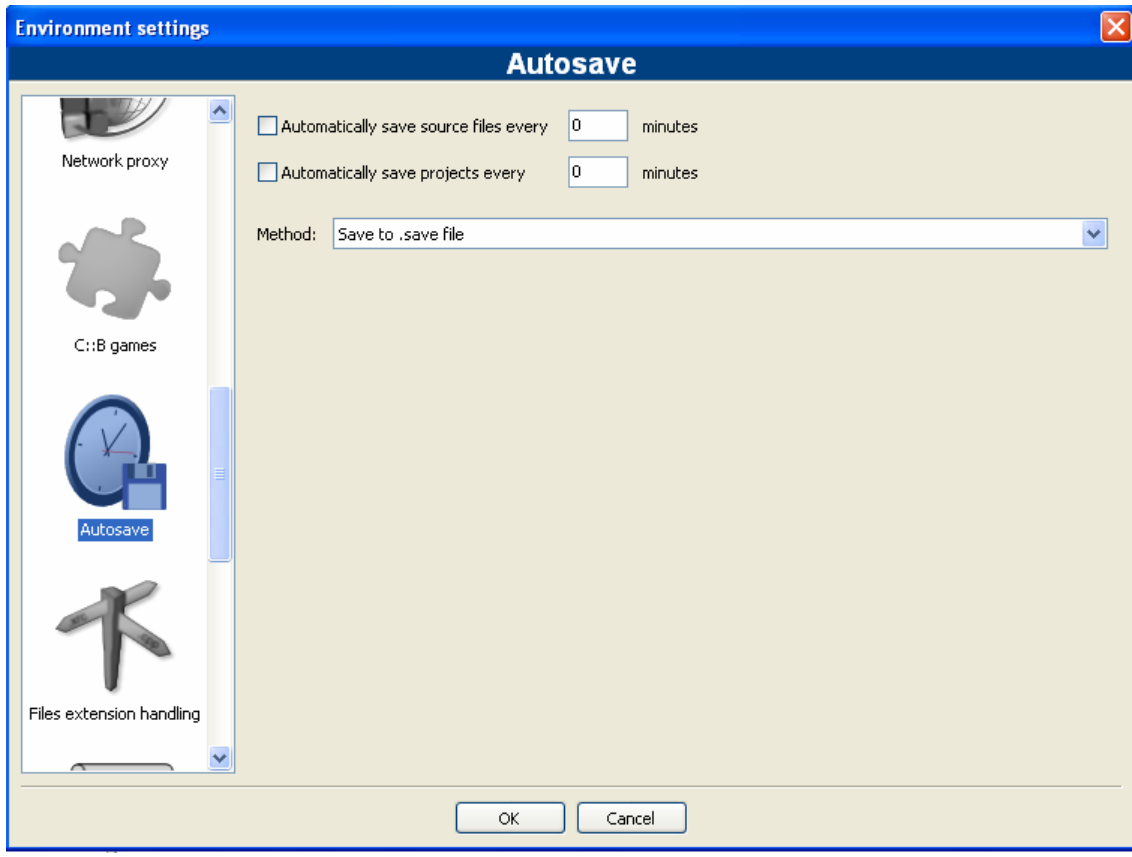
编写或者调试程序的过程中偶尔出现断电，如果没有后被电源，此时可能会丢失部分程序内容。为

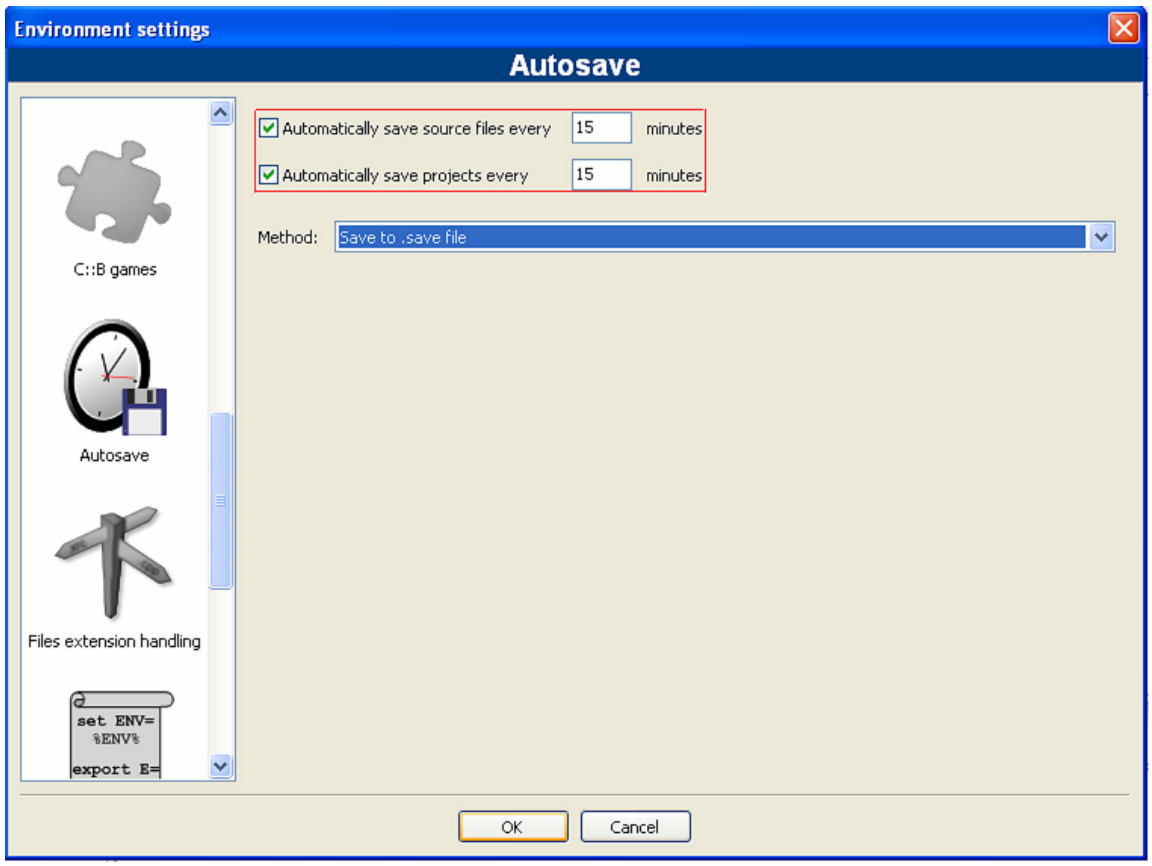
此，我们需要设置Code::Blocks能自动保存功能所对应的选项。

进入Code::Blocks后，选择主菜单Settings下Environment…子菜单，弹出一个对话框，用鼠标拖动左侧的滚动条，找到如右图标。



选中它，界面如下图。





分别设置自动保存源文件和工程的时间，例如均为15分钟，见上图中红色框中部分。

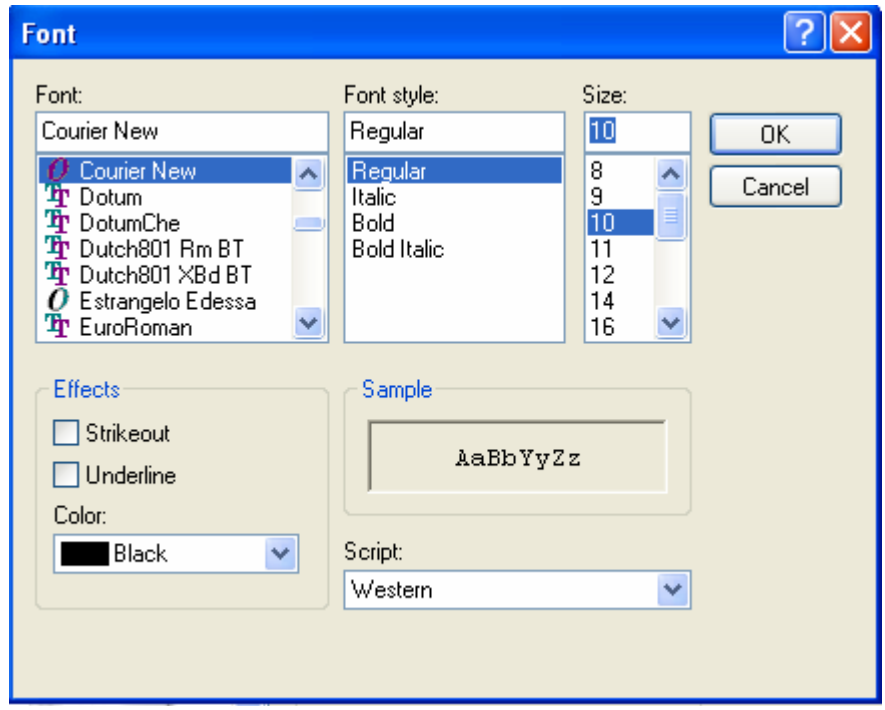
Method为保存文件的方法，有三种，分别是Create backup and save to original file, Save to original file, 以及Save to .save file。选择最后一个，Save to .save file就可以了，设置完毕后，选择OK按钮。

2.2 编辑器

编辑器主要用来编辑程序的源代码，Code::Blocks内嵌的编辑器界面友好，功能比较完备，操作也很简单。

2.2.1 通用设置

启动Code::Blocks，选择主菜单Settings下的子菜单Editor…会弹出一个对话框，默认通用设置General settings栏目，选中一些选项如右下图。



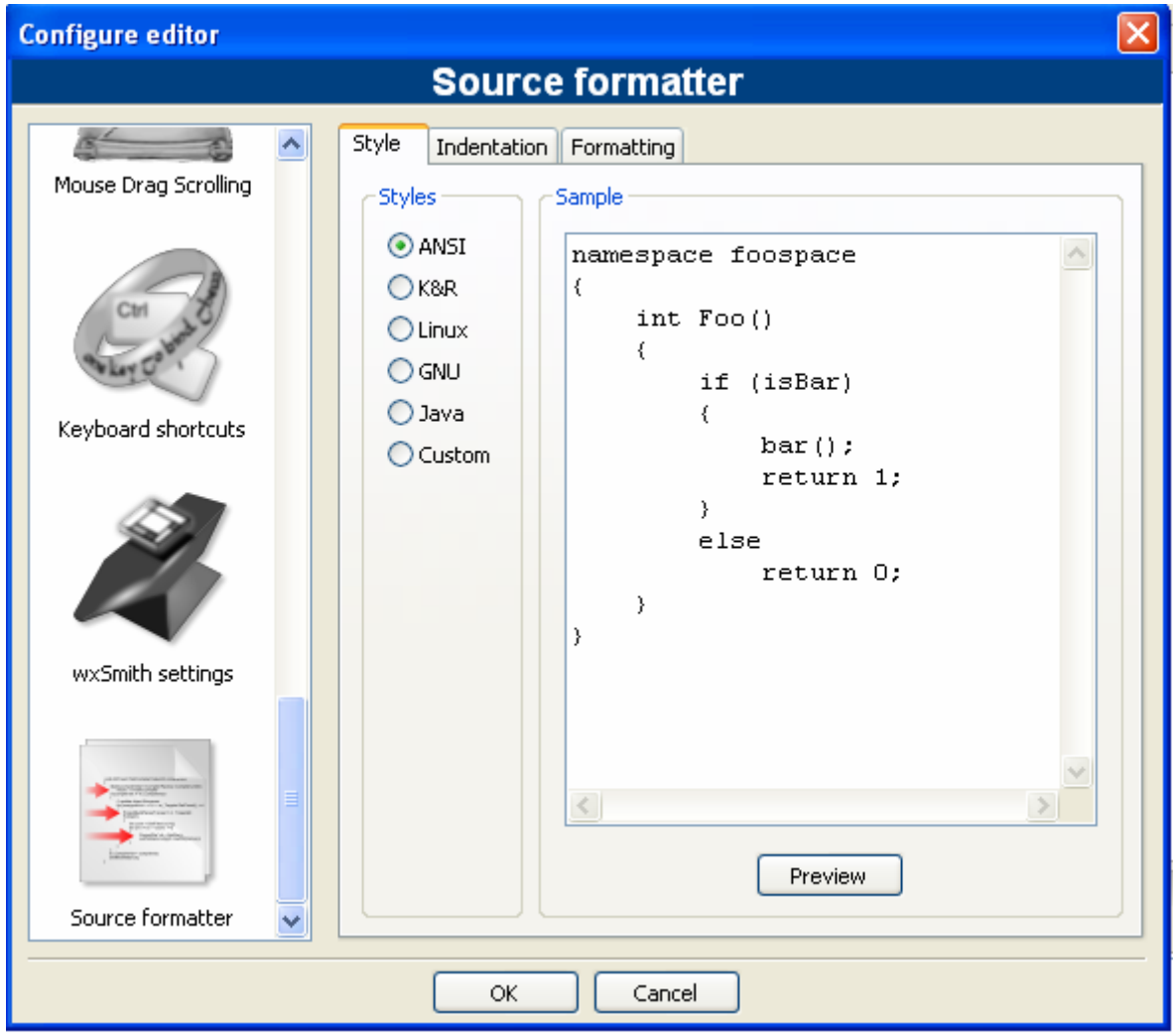
然后设置字体，字体设置首先选择右上角的Choose按钮，会弹出一个对话框，对话框主要有三个竖向栏目，最左侧的栏目Font:用来选择字体类型，选择Courier New，中间栏目Font style: 是字体样式，选择Regular, 最右边的栏目Size:是文字大小，根据个人习惯和电脑显示器显示面积大小进行选择，一般10~12，其它选项不变。见右图。

然后用鼠标选择OK，则字体参数设置完毕，进入上一级对话框General settings，再选择OK，则General settings设置完毕，回到Code::Blocks主界面。

2.2.2 源代码格式

不同的人编写代码风格不同，Code::Blocks提供了几种代码的书写格式。首先从Settings主菜单进入子菜单Editor…，然后从弹出的对话框中移动滚动条，找到标签为Source formatter的按钮，见右图。选中它，可以看到右侧Style菜单下有几种风格分别为ANSI, K&R, Linux, GNU, Java, Custom, 最右侧则是这些风格的代码预览Preview。可以根据个人习惯进行选择，如果选择Custom则需要自己设置两个子菜单Indentation和Formatting下的各个选项，选中自己习惯或者喜欢的风格(笔者的习惯是用ANSI)，然后点击OK按钮。


Source formatter



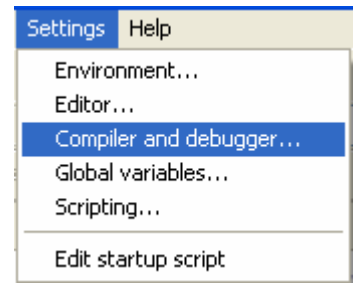
如此以来，编辑器的基本设置就完成了，尽管还有很多其它的选项和参数，但是并不太常用，因此这里就不做详细介绍了。

2.3 编译器和调试器

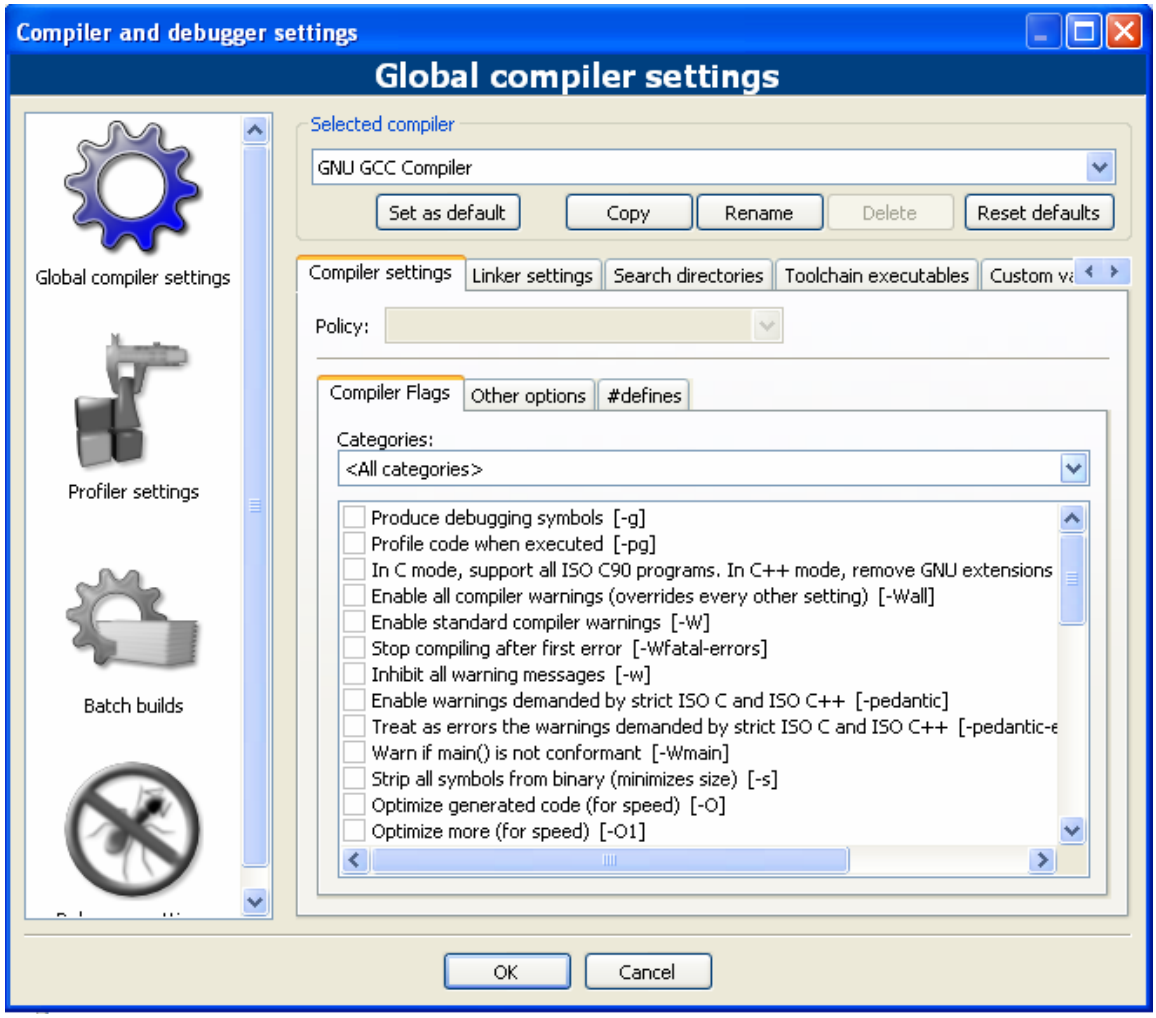
编译器和调试器可能最重要，因为编写的源代码需要转换成机器可以识别的二进制才能执行，而且编写程序时，很难保证一次正确。这里主要讲述一下，编译器和调试器的全局配置，也就是说这里配置的每个选项都会影响到将来建立的每个工程。

2.3.1 进入编译器和调试器配置界面

首先从Code::Blocks界面的主菜单Settings下拉菜单下选择Compiler and debugger…按钮，见右图。

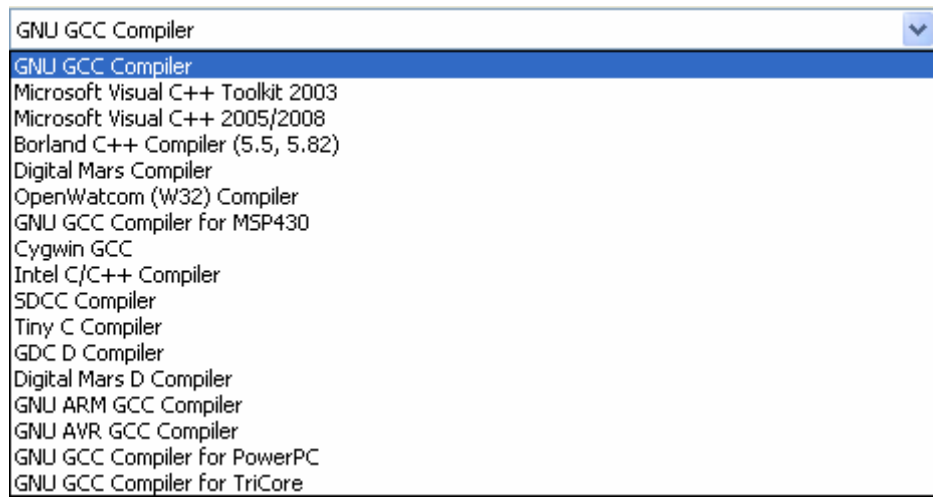


会弹出一个对话框如下，这就是编译器和调试器的配置界面，见下图。



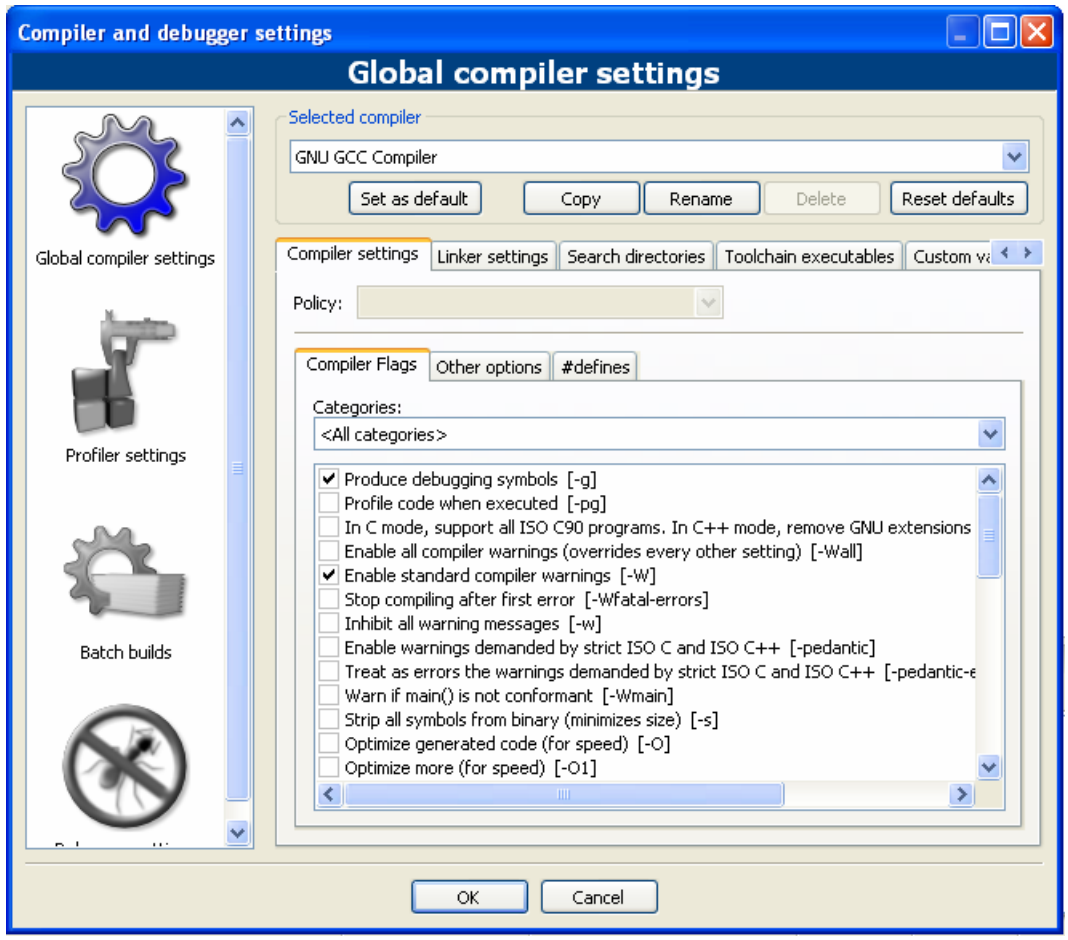
2.3.2 编译器选择

Code::Blocks支持多种编译器，默认编译器GNU GCC Compiler，当然，您也可以选择其它的编译器，只不过需要事先安装好您想用的编译器，见右图。

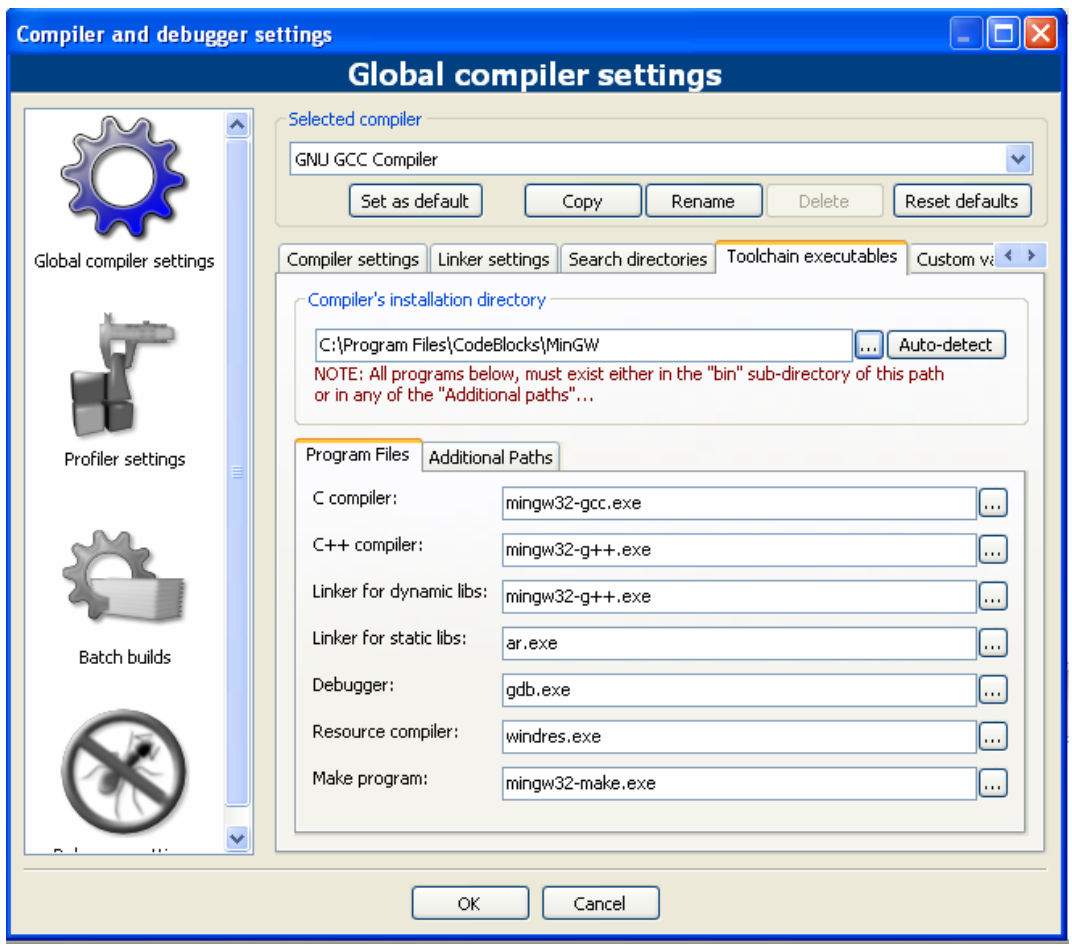


2.3.3 扩展编译选项配置

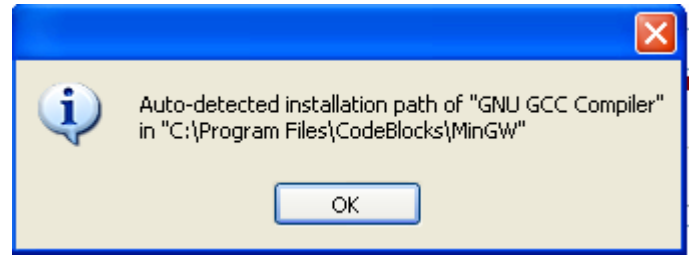
选择Compiler Settings菜单下的Compiler Flags子菜单，选中其中两个选项，Produce debugging symbols [-g]和Enable standard compiler warnings [-W]，也可以什么都不选，见下图。



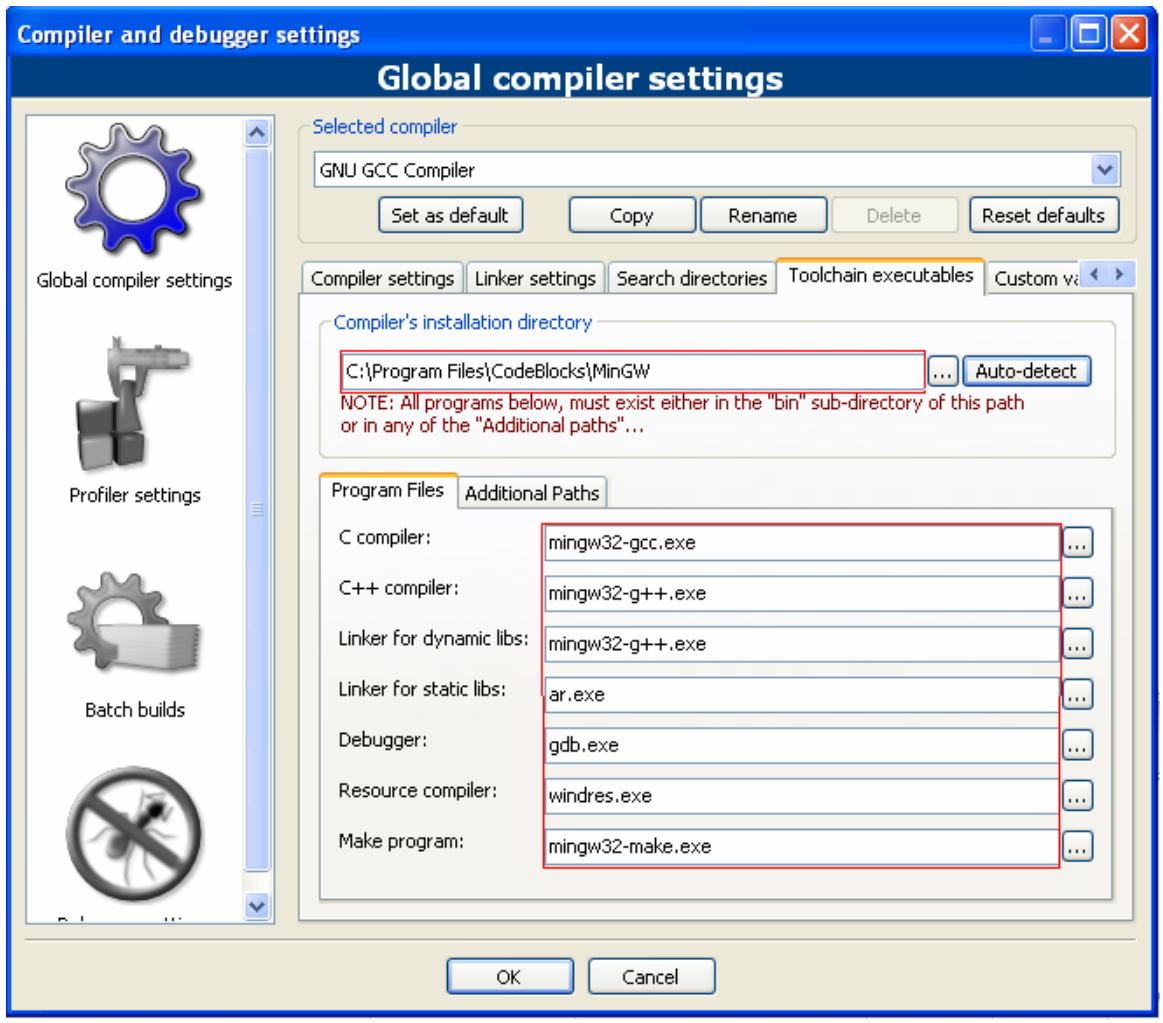
然后选择Toolchain executables子菜单，会出现一个界面，见下图。



点击右侧的Auto-detect按钮，一般而言能自动识别编译器的安装路径，见右图。



如果不能自动识别编译器安装的路径，就需要点击按钮[...]进行手工配置好该路径。并且也要配置好C compiler:, C++ compiler:, Linker for dynamic libs:, Linker for static libs:, Debugger:, Resource compiler:, Make program:这几个选项的文件名。见见下图中用红色方框框起来的部分。

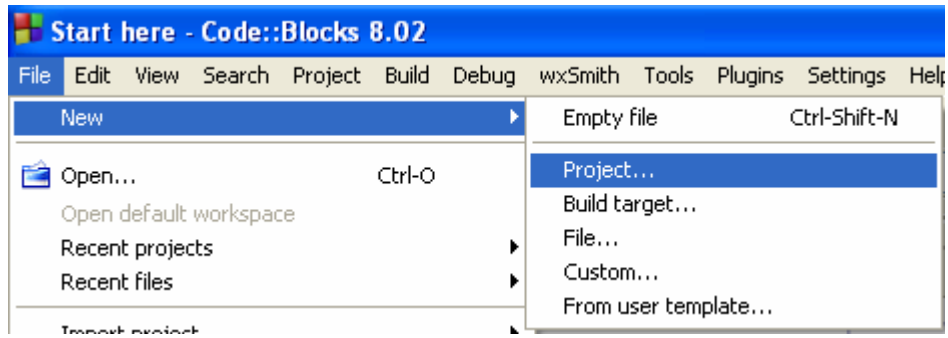


最后用鼠标点击最下方的OK按钮，则编译器和调试器基本配置完毕。

3. 编写程序

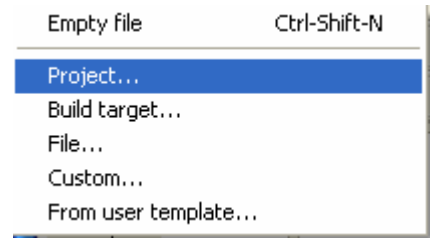
下载安装Code::Blocks以后，您就可以编写代码了。Code::Blocks创建一个工作空间(workspace)跟踪你当前的工程(project)。如果有必要，您还可以在您当前的工作空间创建多个工程。一个工程就是一个或者多个源文件(包括头文件)的集合。源文件(source file) 就是您程序中包含源代码的文件，如果您正在编写C++程序，您就正在编写C++源代码(文件后缀名为.cpp)。您创建库文件(library files, 文件后缀名为.h或.hpp)时，会用到头文件(header file) 。一个库(library)是为了实现特定目标的函数集合，例如数学运算。

创建一个工程可以方便的把相关文件组织在一起。一个工程刚建立时，一般仅仅包含一个源文件。但是，伴随着您编程经验的增长可能会用到更复杂的工程，此时一个工程可能包含很多源文件以及头文件。



3.1 创建一个工程

创建工程的方法很多，您可以选择主菜单File的下拉菜单中选择二级菜单New，然后从子菜单中选择Project…，如左上图。

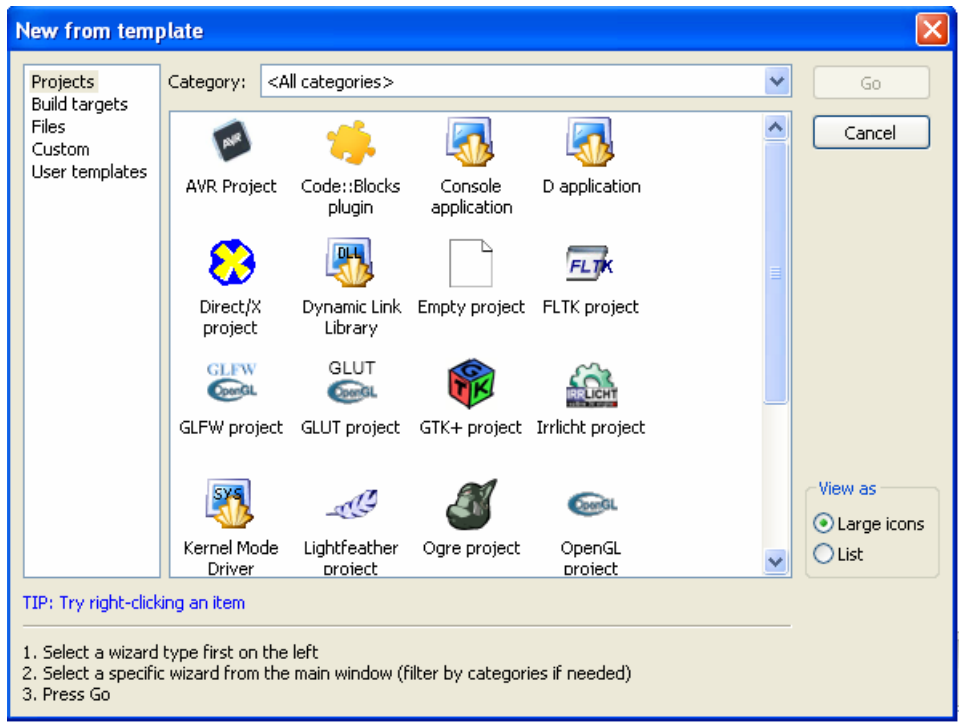


也可以从图标按钮 开始创建，用鼠标点击File下面的 ，会弹出一个对话框，从弹出的对话框中选择Project…按钮，如右图：

您还可以从Code::Blocks主界面中选择Create a new project按钮进行创建，见右面Create a new project的图标。



无论使用哪种方式创建一个工程，都会打开一个对话框，见下图。

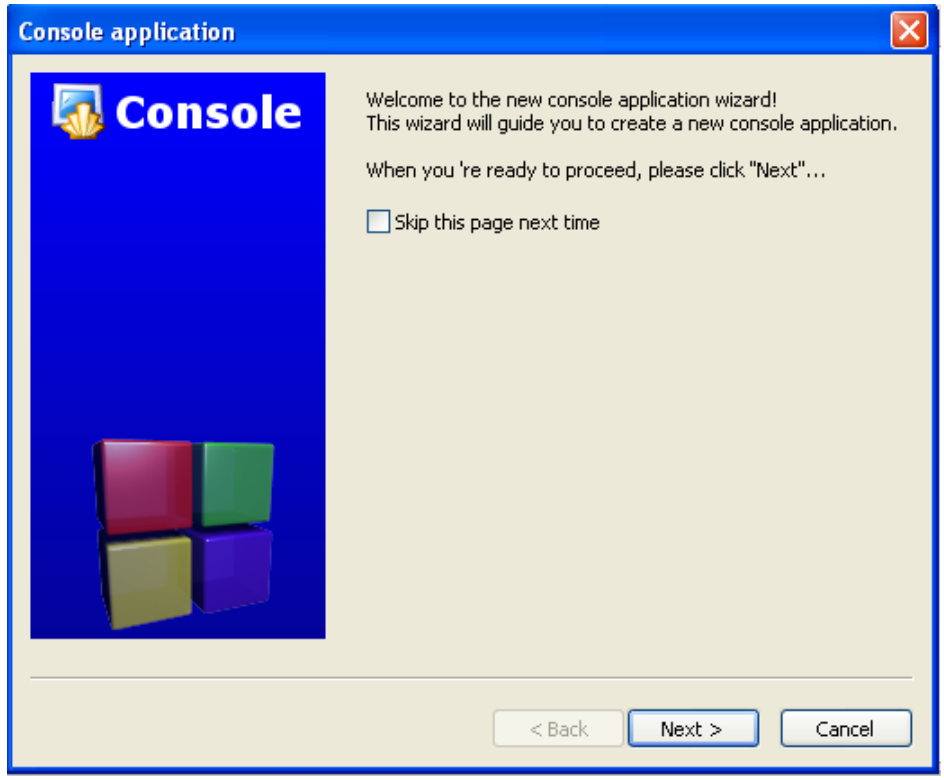


这个窗口中含有很多带有标签的图标，代表不同种类的工程。我们最常用的可能是Console application，用来编写控制台应用程序。其它的是一些更高级的应用。

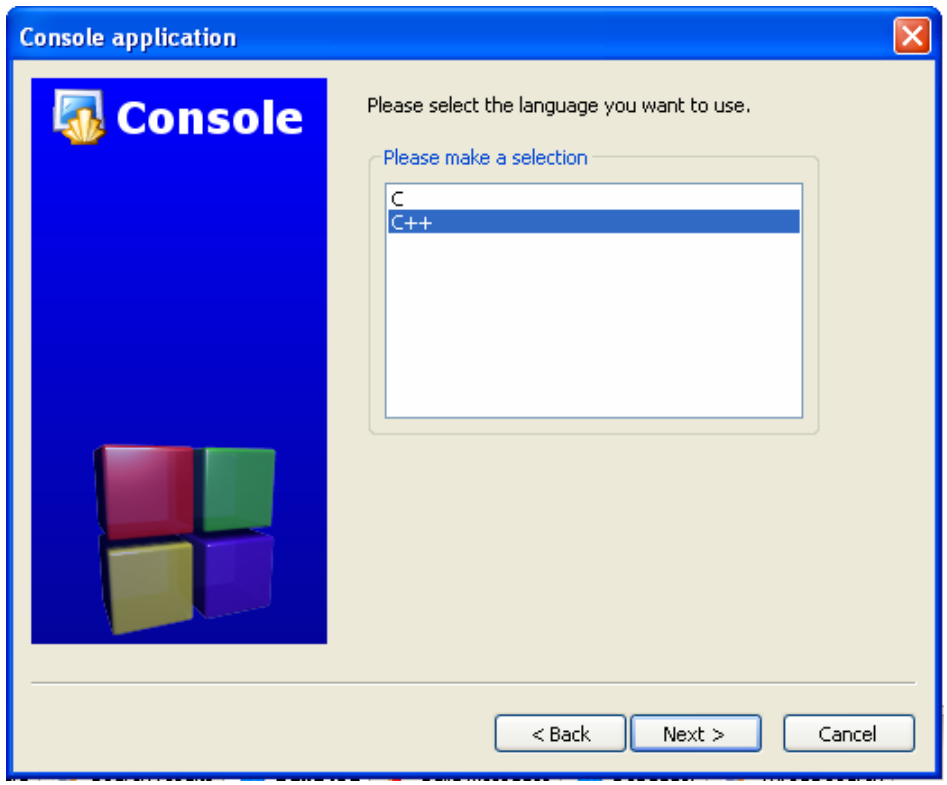
用鼠标选中带有控制台应用(Console application)标签的图标，见右图。



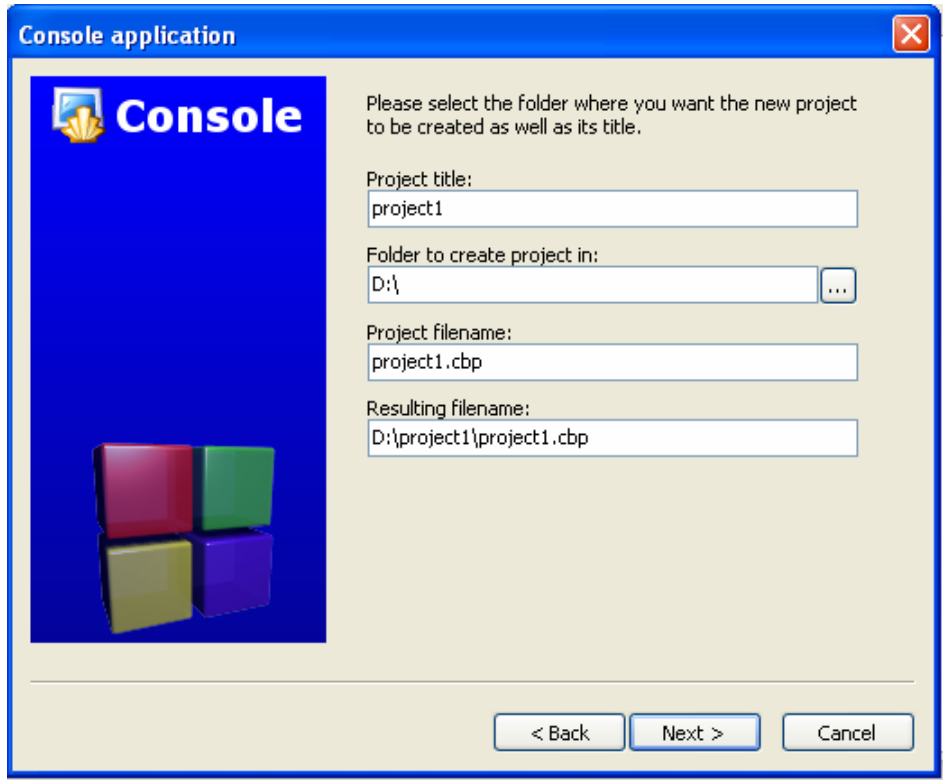
再选择右侧的Go按钮，这样会弹出一个新的对话框，见下图。



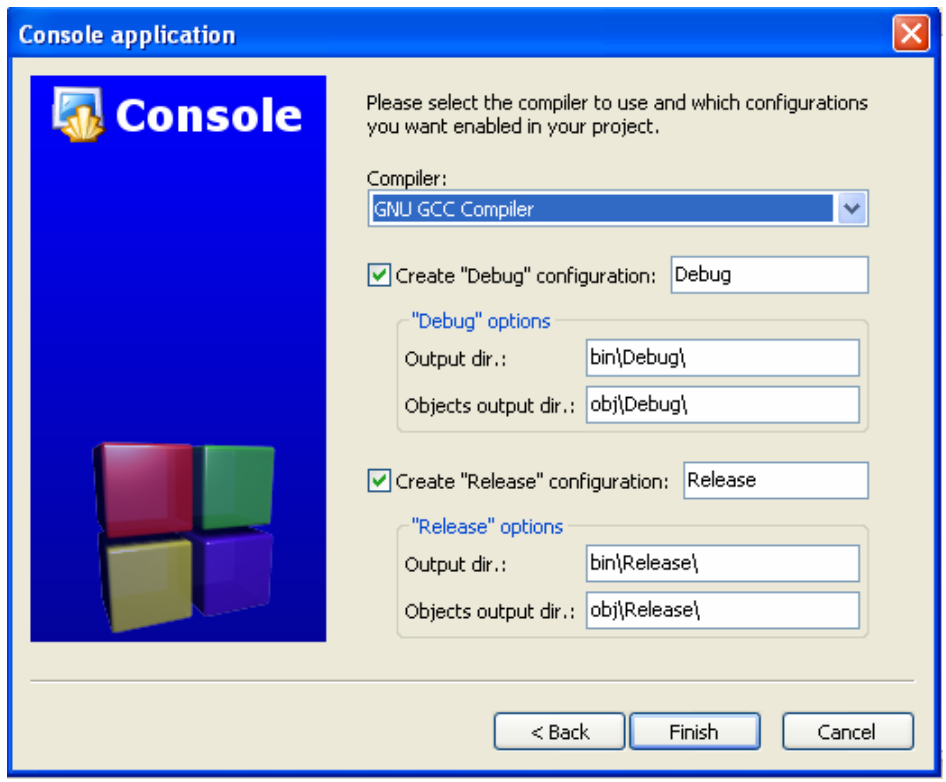
再选择Next按钮进入下一步，弹出一个对话框，见下图。



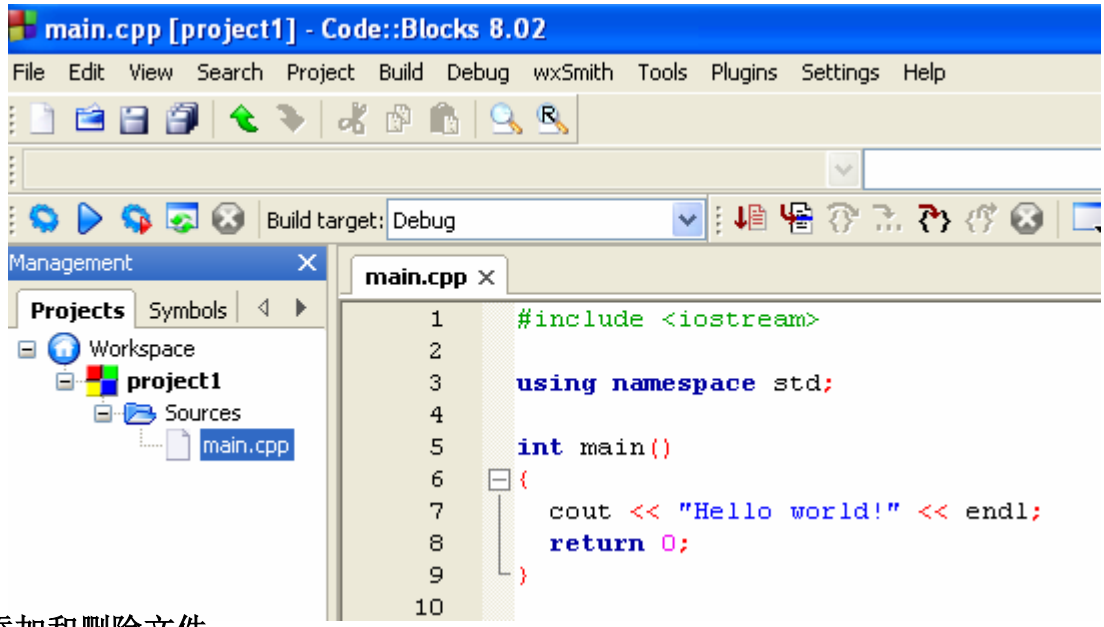
在弹出的对话框中有C和C++两个选项，选择C++表示编写C++控制台应用程序，选择C表示编写C控制台应用程序。这里以编写C++程序为例，因此选择C++，见上图。接下来选择下方的Next按钮进入下一步，又弹出一个对话框，见下图。



弹出的对话框中有4个需要填写文字的地方，填上前两个(工程名和工程文件夹路径)，后两个位置需要填写的内容可以自动生成。见上图。然后选择Next按钮进入下一步，见下图。



编译器选项仍旧选择默认的编译器，剩下的全部打勾，见上图。然后选择Finish按钮，则创建了一个为project1的工程。用鼠标点逐级击⊞使之变成⊟，依次展开左侧的project1，Sources，main.cpp，最后显示文件main.cpp的源代码，见下图。

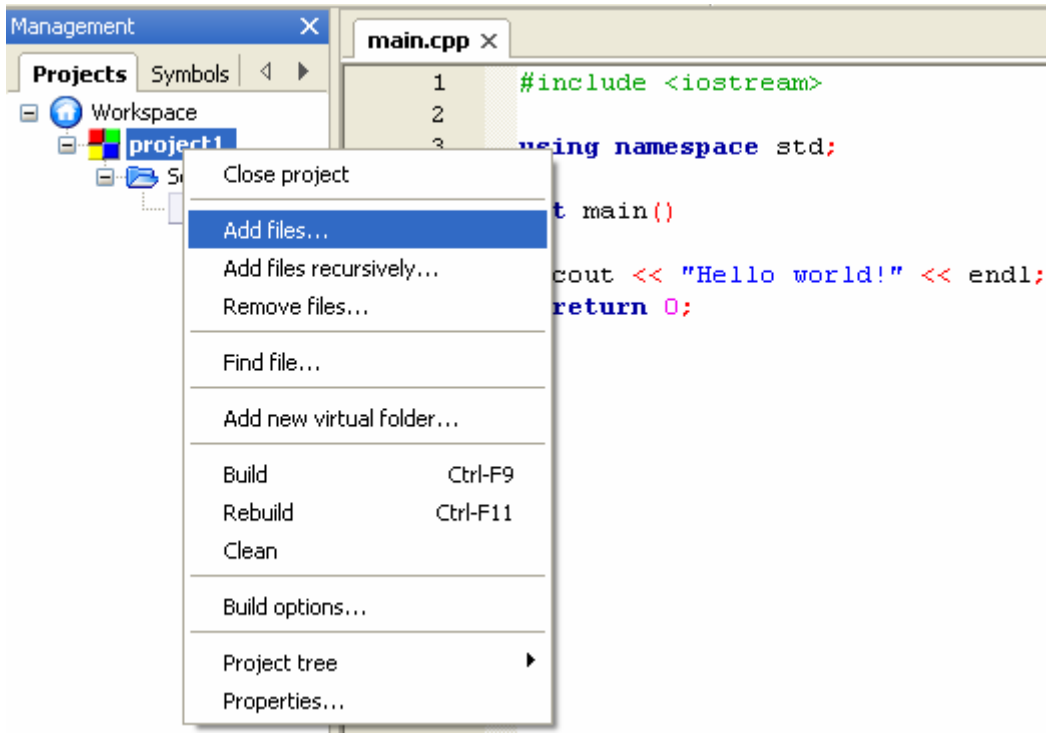


3.2 添加和删除文件

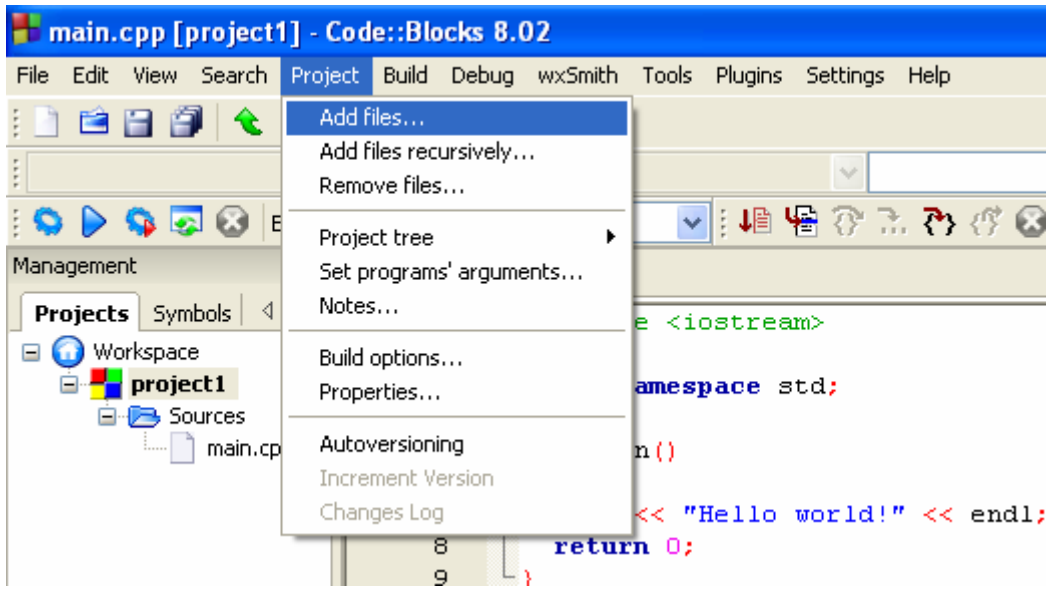
当建立一个工程后，我们往往需要往工程中添加新文件，工程不需要的文件则要从工程中删除之。

3.2.1 添加文件

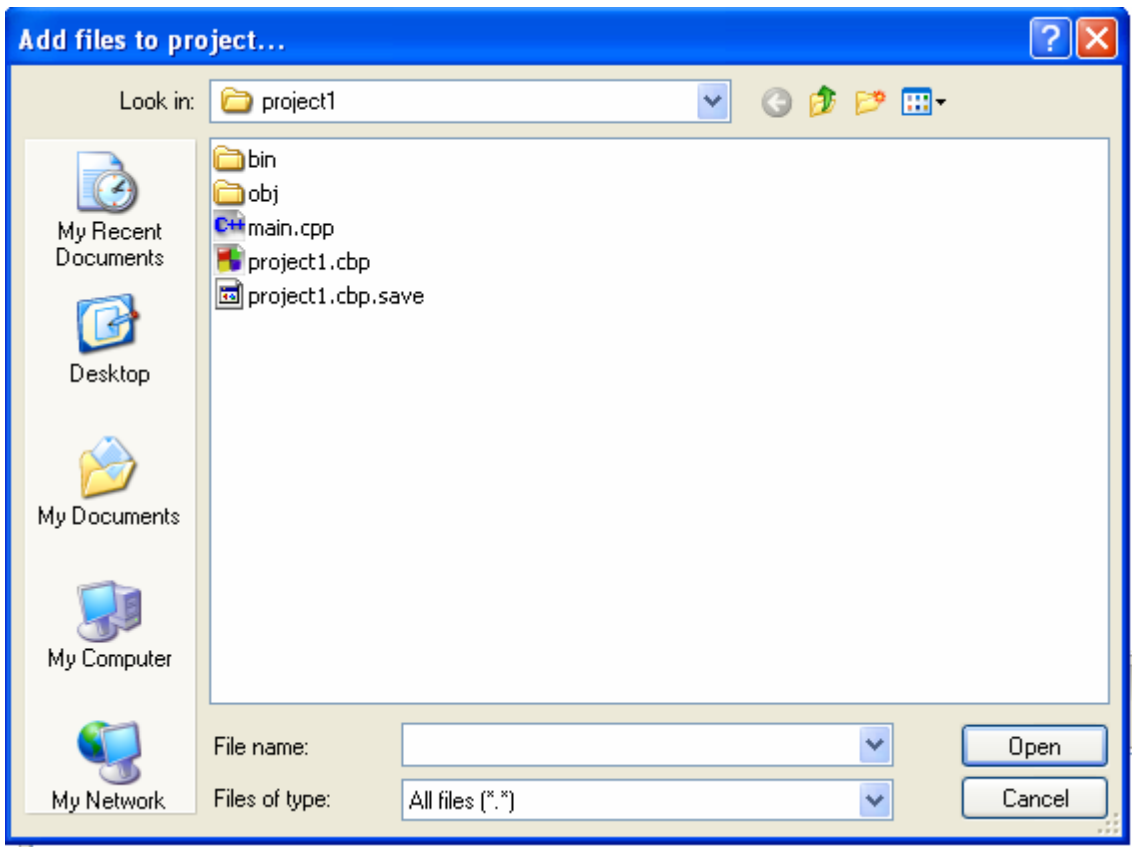
给工程添加文件的方法很多，一种办法是移动鼠标选择刚建立的工程题头上(project1)，按下鼠标右键，弹出一个菜单，菜单上有几个按钮，Close project是关闭当前工程，Add files…是添加文件到工程中，另外还有Remove files…是从当前工程中删除文件。Build用来编译当前工程，Rebuild用来重新编译当前工程，Clean用来清除编译生成的文件，见下图。



另外一种方法是从Project主菜单选择下拉菜单中的按钮。Add files…用来添加新文件到当前工程中，Remove files…是从当前工程中删除文件，见下图。

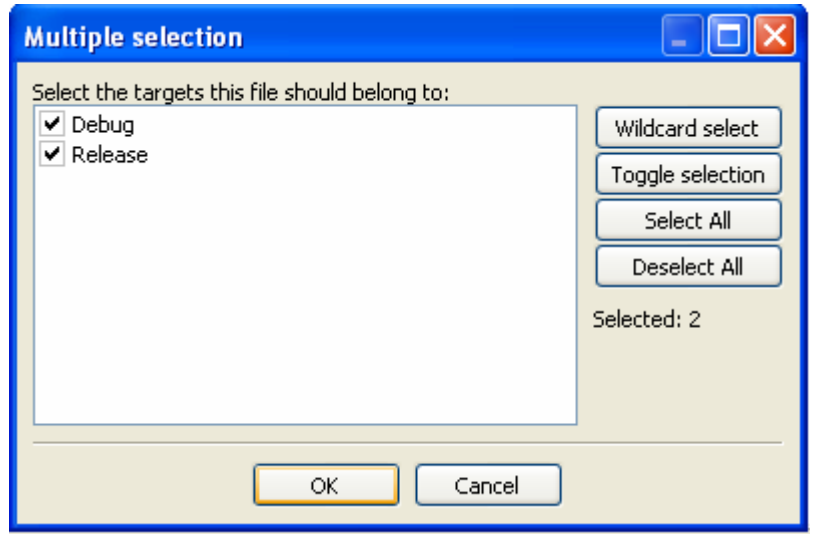


用鼠标点击上面介绍的菜单中Add files…按钮，会弹出一个对话框，见下图。

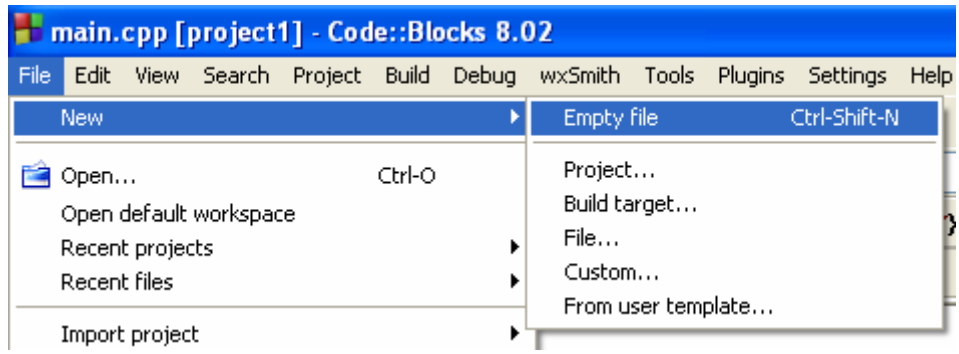


选择要添加的文件，然后选择右下角的Open按钮，见上图。

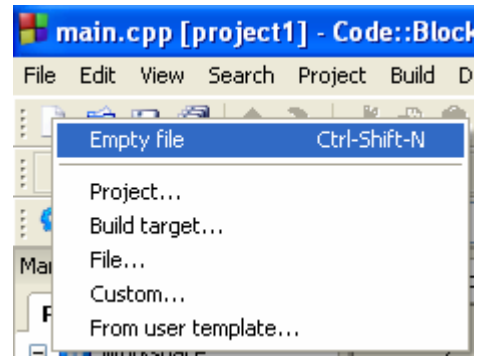
点击Open后，又弹出一个对话框见右图，把Debug和Release全部选中，然后选择OK按钮，则添加到了当前中。



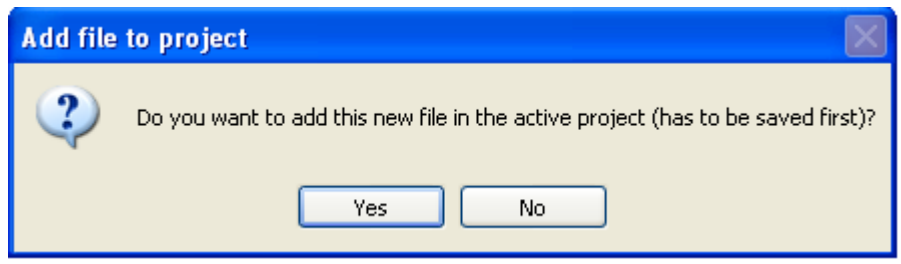
如果此时存储设备上没有我们需要的文件，则需要自己创建一个。创建新文件的方法很多，可以选择File菜单，从下拉菜单中中选择New，通过向右的导向箭头打开New的子菜单，再从这个下拉菜单中选择按钮Empty file Ctrl-Shift-N，见右图。



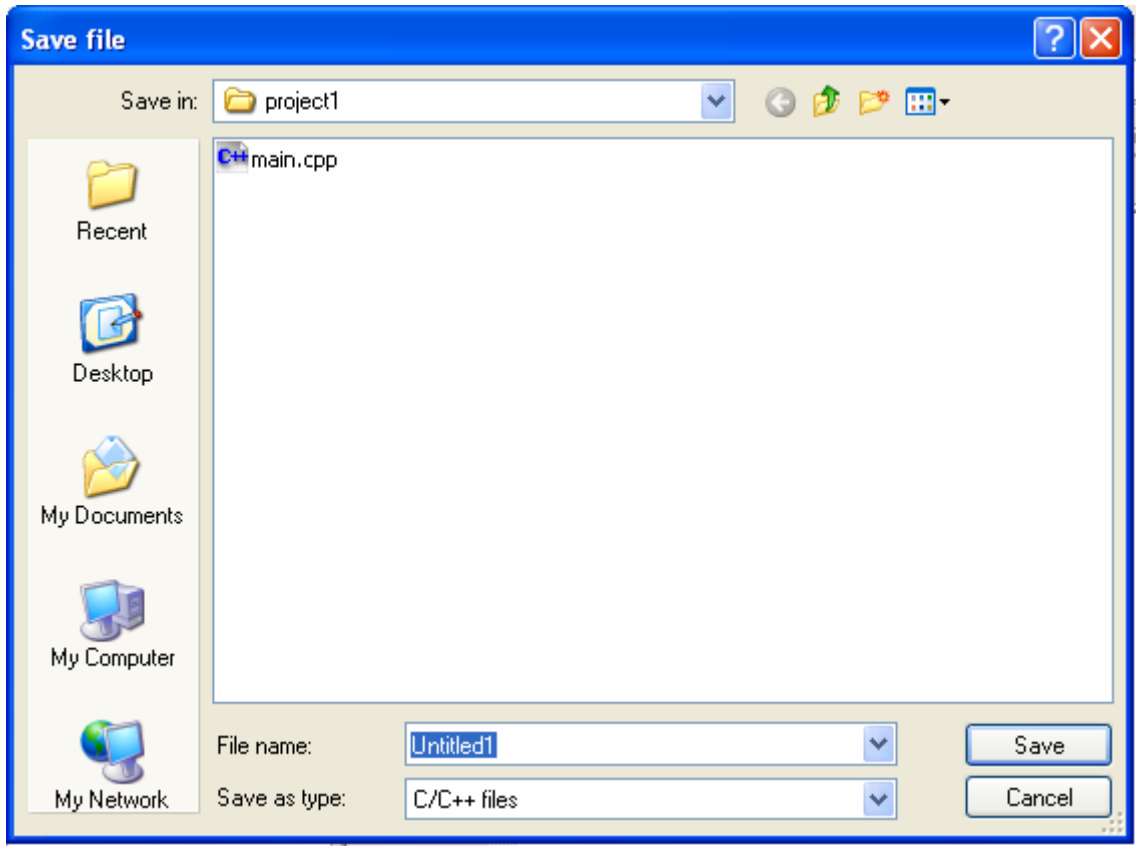
也可以选择这个图标，用鼠标点击一下，也会弹出一个菜单，从弹出的菜单中选择Empty file Ctrl-Shift-N，见右图。



如果您建立了一个工程，然后又想新建一个空文件，则Code::Blocks会问您是否要把这个新文件添加到当前处于激活状态的工程中，见下图。

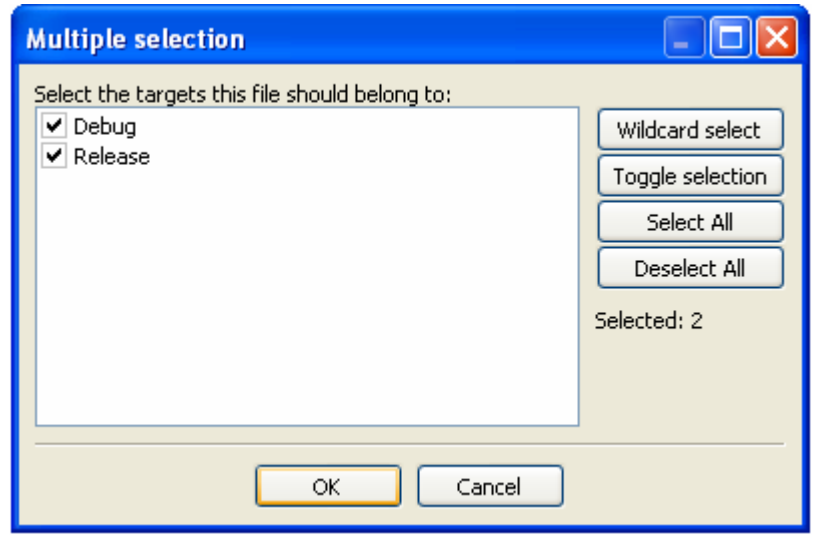


如果您选择Yes，则该文件会被添加到工程中，选择No，则此文件不会被添加到工程。如果您选择了Yes，则会弹出一个新的对话框，让您给新建的这个文件命名，见下图。

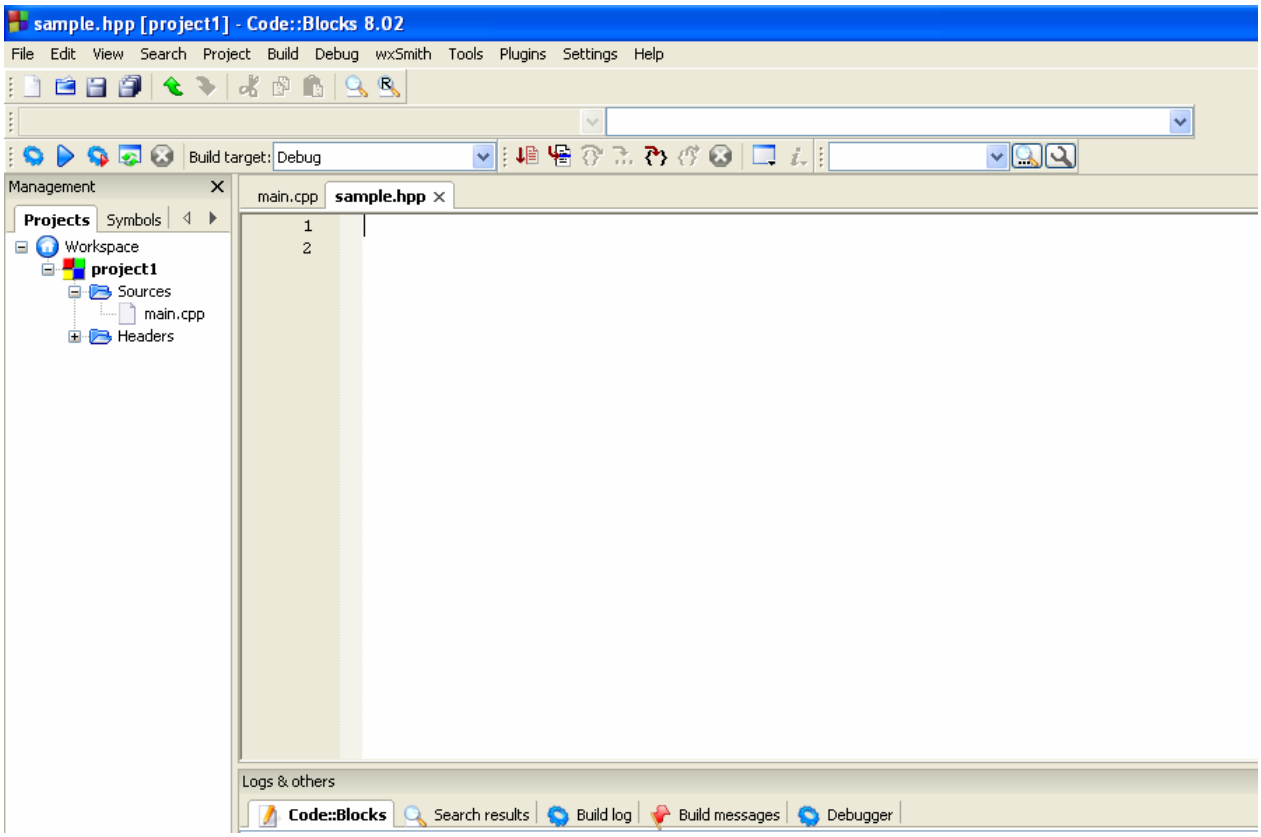


我们假设把这个文件取名为sample.hpp，把它保存到新建立的project1文件夹下面。

当用鼠标点击Save后，又弹出一个对话框，问您目标(target)文件属于哪种类型，选中Debug和Release，然后选择OK。见右图。



这样我们就可以编辑sample.hpp了(文件名后缀为.hpp或者.h的是头文件)，系统自动把它归为头文件，见下图。

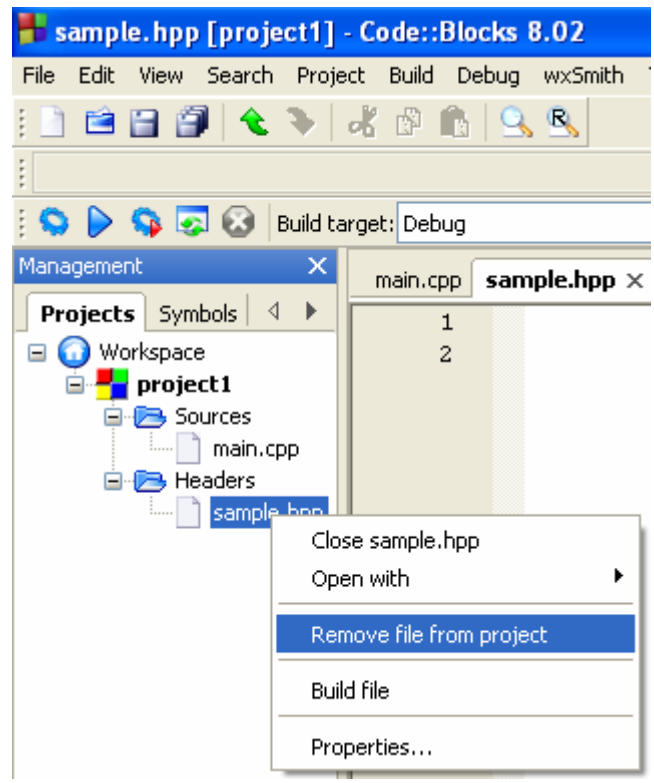


一个目标文件是编译后的文件，可以为debug或者release，debug版本的目标文件允许您使用调试器对该文件进行测试。一般而言，debug版本的目标文件通常较大，因为它包含了一些用于测试的额外信息，release版本的目标文件一般较小，因为它不包含调试信息。当您的程序编译完毕，应该交付release目标文件。

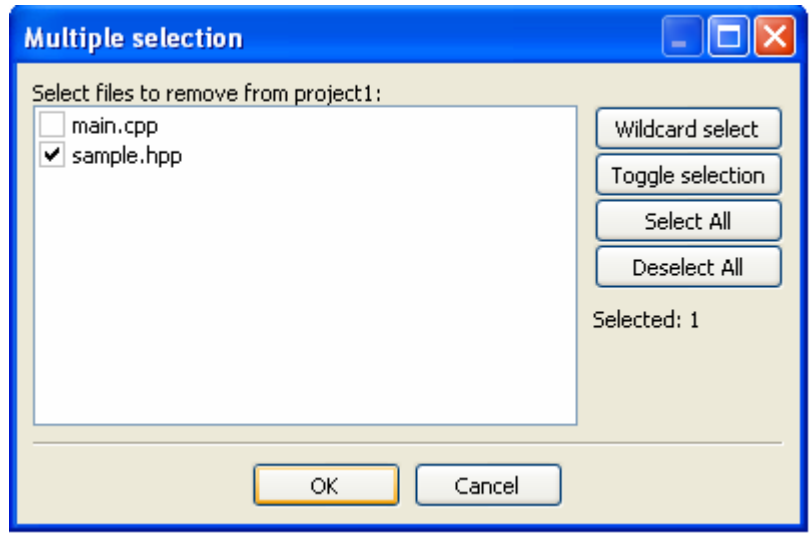
3.2.2 删除文件

我们前面创建的文件sample.hpp仅仅是为了示例说明如何创建文件，因此并不是project1所必须的，需要把它从project1中删除。

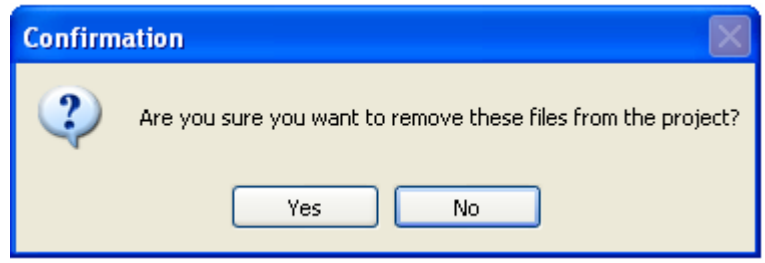
删除这个文件方法很多，这里简要说明两种，一种方法是用鼠标点击图标 Headers 上的，这样Headers就会展开，选择sample.hpp，按下鼠标右键弹出的菜单中选择按钮Remove file from project，则sample.hpp就不在隶属于project1了，见右图。



我们还可以从主菜单选择Project的下拉菜单中选择Remove files…按钮，这样会弹出一个对话框，对话框中是当前处于激活状态工程project1的所有文件，需要把哪个文件从该工程中删除，则选中那个文件，然后选择OK。见右图。

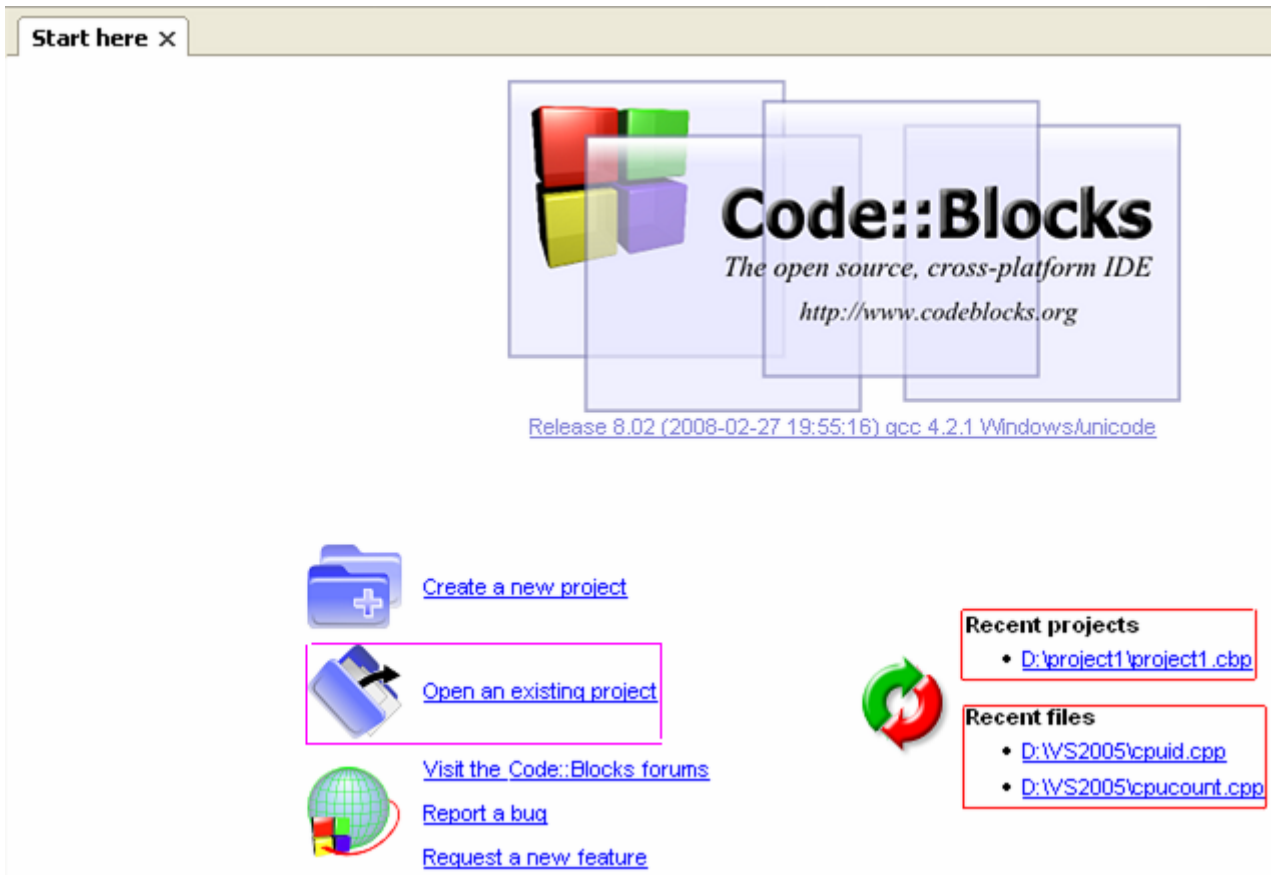


当用鼠标点击OK按钮后，还会弹出一个对话框让您确认，选择Yes，则此文件sample.hpp就不再隶属于当前工程project1了，见右下图。

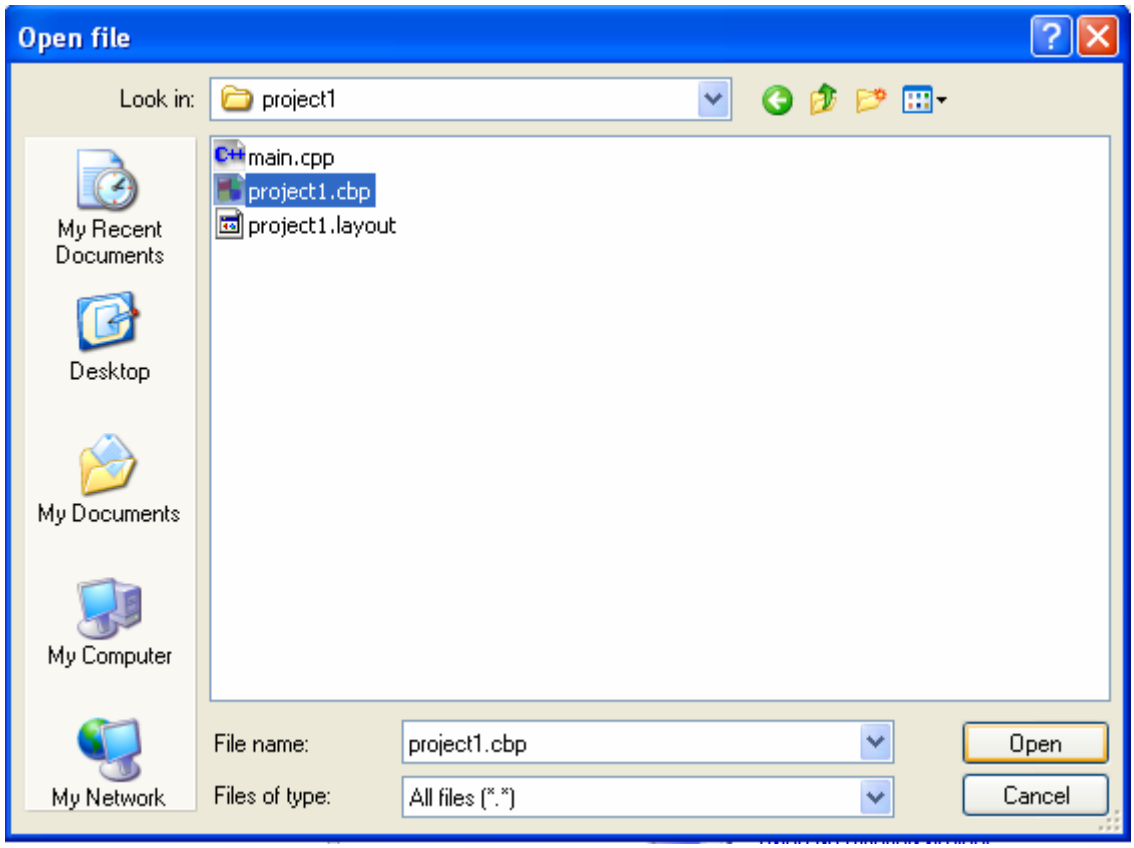


3.3 编辑文件

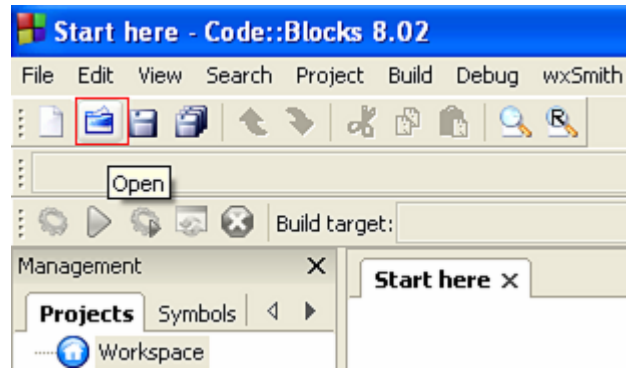
编辑已经存在的文件前需要首先打开这个文件。打开文件的方法很多，由于Code::Blocks会默认记住打开的文件路径，进入Code::Blocks主界面后可以看到最近用Code::Blocks打开过的工程以及最近打开过的文件。见右下图中图标右侧用红色方框框起来的部分Recent projects以及Recent files。用鼠标选择之，即可打开对应的工程或文件。



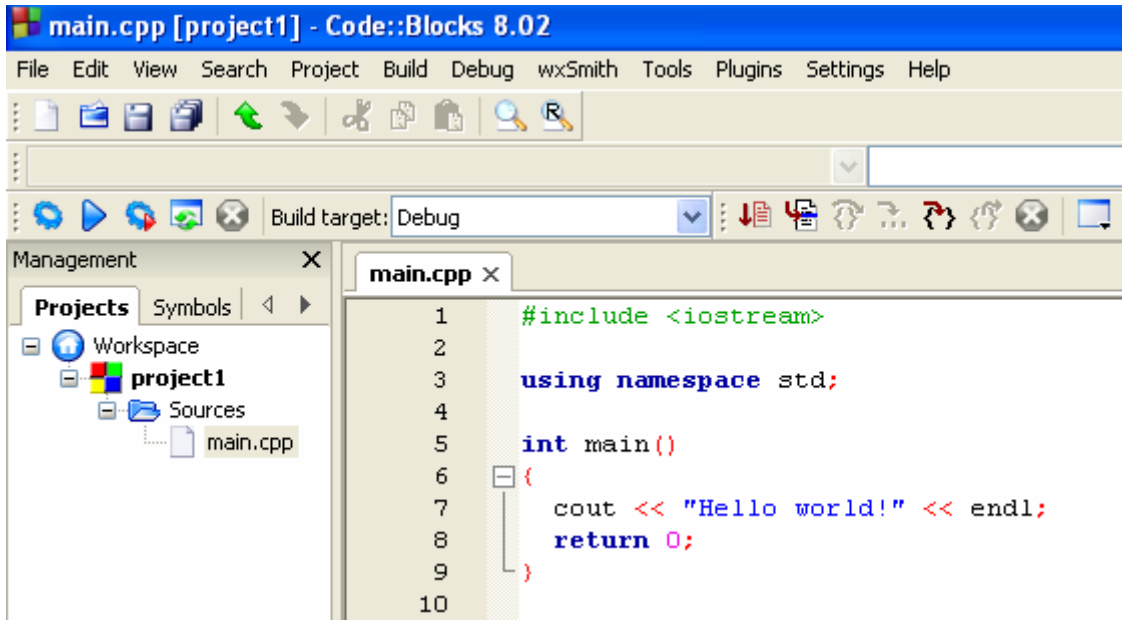
也可以选择带有标签Open an existing project的图标，见右图中用粉红色方框框起来的部分，用鼠标点击它则会弹出一个对话框见右下图，让您选择要打开的工程，选中对应文件名，点击Open按钮。



还可以从主界面Edit主菜单下面的图标，见左下图。点击该图标会弹出一个对话框，对话框见右图，然后找到要打开的文件，选中它，点击Open按钮，也可以打开相应的工程或者文件。

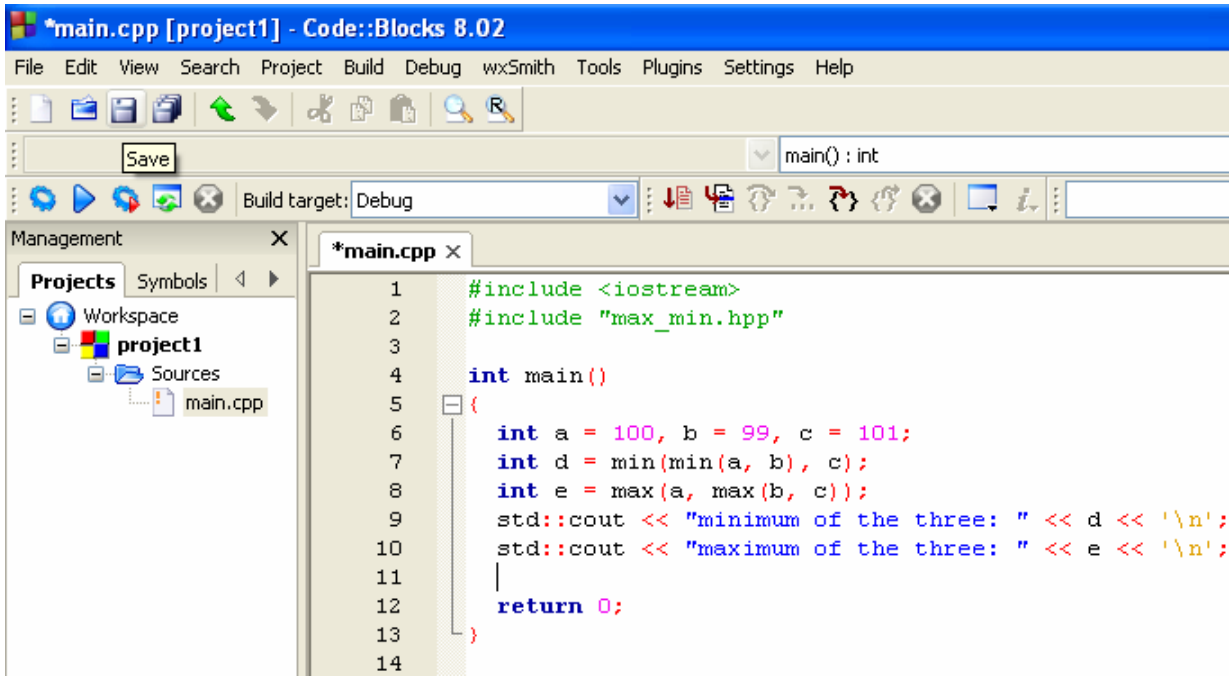


打开一个工程后再打开工程中的某个文件，则显示该文件的源代码，见下图。

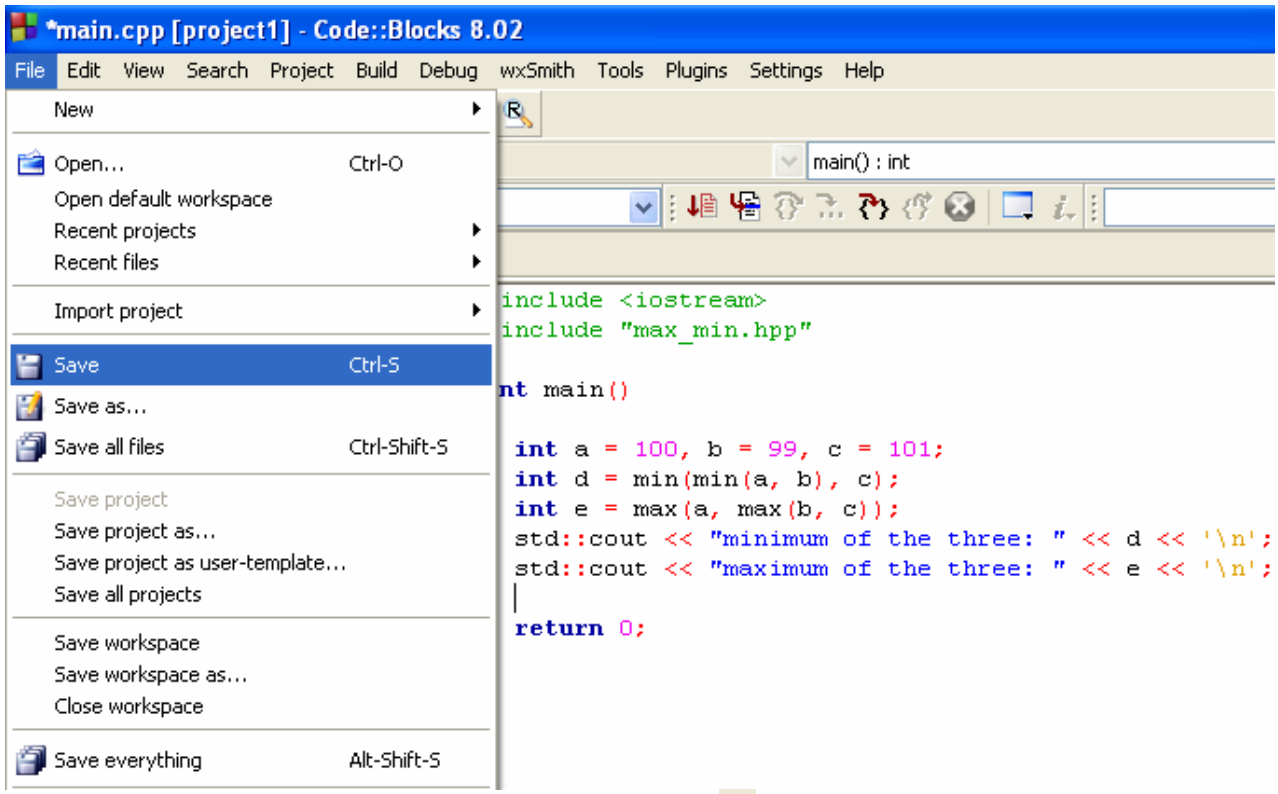


假设我们想修改此文件的源代码，让它求出三个整数中的最小值，则可以这样做。首先，我们修改main.cpp的源代码，修改完毕后，我们保存当前文件。

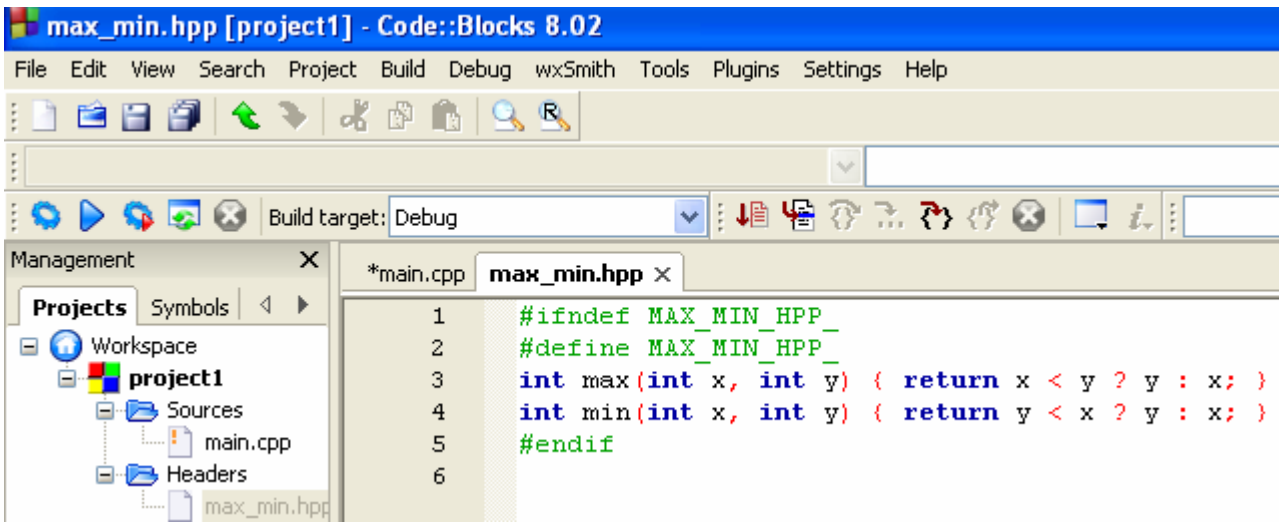
保存当前文件方法很多，常见的有两种，一种是用鼠标点击一下按钮，见下图。



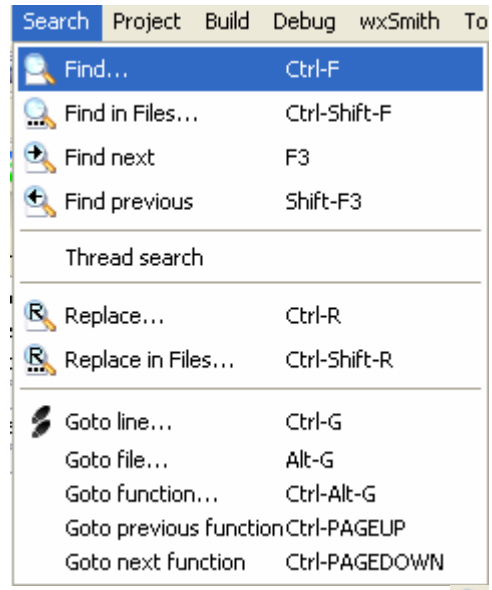
另外一种方法是在主菜单File的下拉菜单中选择按钮 Save Ctrl-S，见下图。



假如我们想保存project1中的所有文件，可以找到图标，用鼠标点击它一下即可，或者从File的下拉菜单中找到按钮 Save all files Ctrl-Shift-S，用鼠标点击一下也可。如果我们想保存整个project1工程，也可以从File的下拉菜单中找到按钮 Save project 点击一下则就保存了project1。此外File的下拉菜单中还有 Save all projects按钮，用来保存当前工作空间中的所有工程，Save workspace按钮用来保存当前工作空间， Save everything Alt-Shift-S按钮用来保存Code::Blocks已经打开的所有内容。如果我们想给当前文件创建一个副本，改成其它名字则在File的下拉菜单中选择 Save as…按钮，取一个新名字保存之，如果我们想给当前工程创建一个副本，改成其它名字则在File的下拉菜单中选择 Save project as…按钮，取一个新名字保存之。此外还有Save workspace as…等。如果想关闭当前文件选择File下拉菜单中的按钮 Close file Ctrl-W，关闭当前工程选择按钮 Close project 。关闭所有文件就选择Close all files Ctrl-Shift-W按钮，关闭所有工程选择Close all projects 按钮。



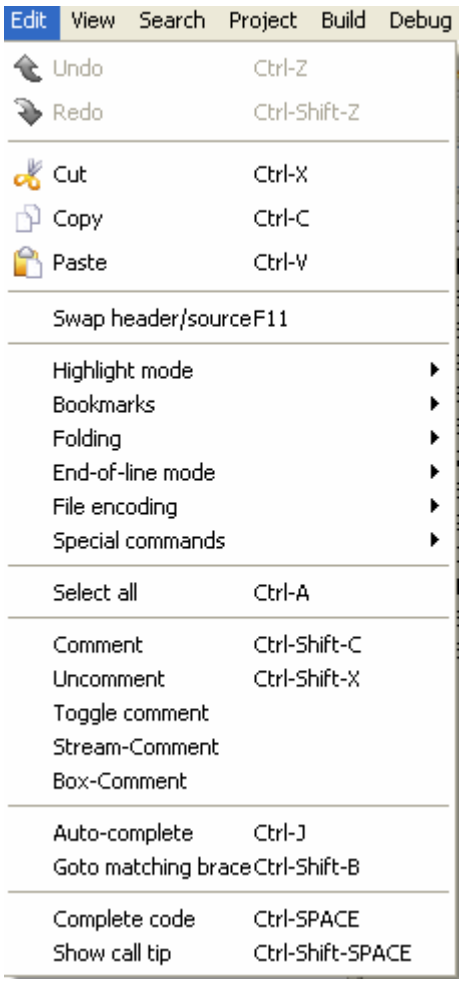
注意到，编辑后的main函数源代码中添加了一行#include “max_min.hpp”，max_min.hpp是一个头文件，该头文件包含了两个函数，一个是max，一个是min，分别用来求出两个整数的较大和较小者。现在我们需要建立这个头文件，并把它添加到project1中。添加文件到工程的方法前面已经有所论述，在此不再重复，假设我们已经编辑完毕max_min.hpp头文件，并添加到project1中，如上图。

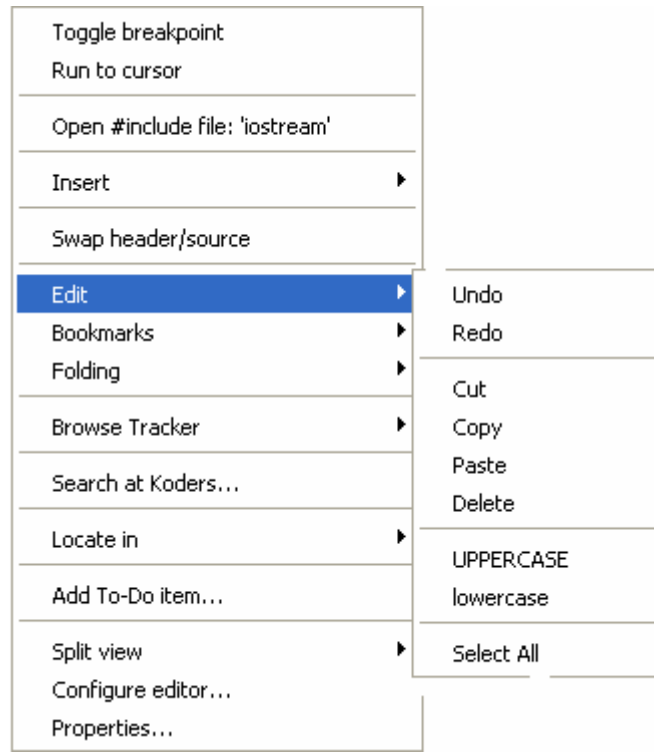


如果编写的源代码较多，有时候可能需要查找和替换某个字符窜。Search主菜单的下拉菜单中有几个按钮用来对文件内容进行操作，可以查找和替换等，见右上图。



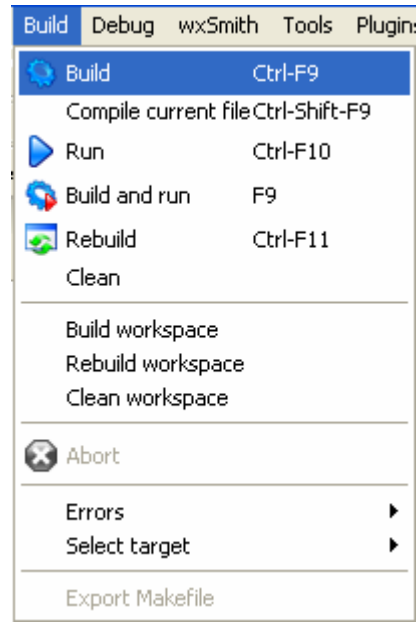
此外，还有一些快捷按钮，可以完成查找(Find)或替换操作(Replace)，见上图用红色方框框起来的两个按钮。该快捷菜单上还有其它功能按钮，Undo(向前反悔), Redo(向后反悔), Cut(剪切), Copy(复制), Paste(粘贴)等，操作简单实用。Edit下拉菜单提供了更加丰富的功能按钮，除了上述这些外，还有Comment(注释)，去掉注释(Uncomment)，Folding(折叠)，File encoding(文件编码选项) 等等，见左下图。此外，编辑模式下，按下鼠标右键弹出的快捷菜单中也有很多编辑选项，见右下图。

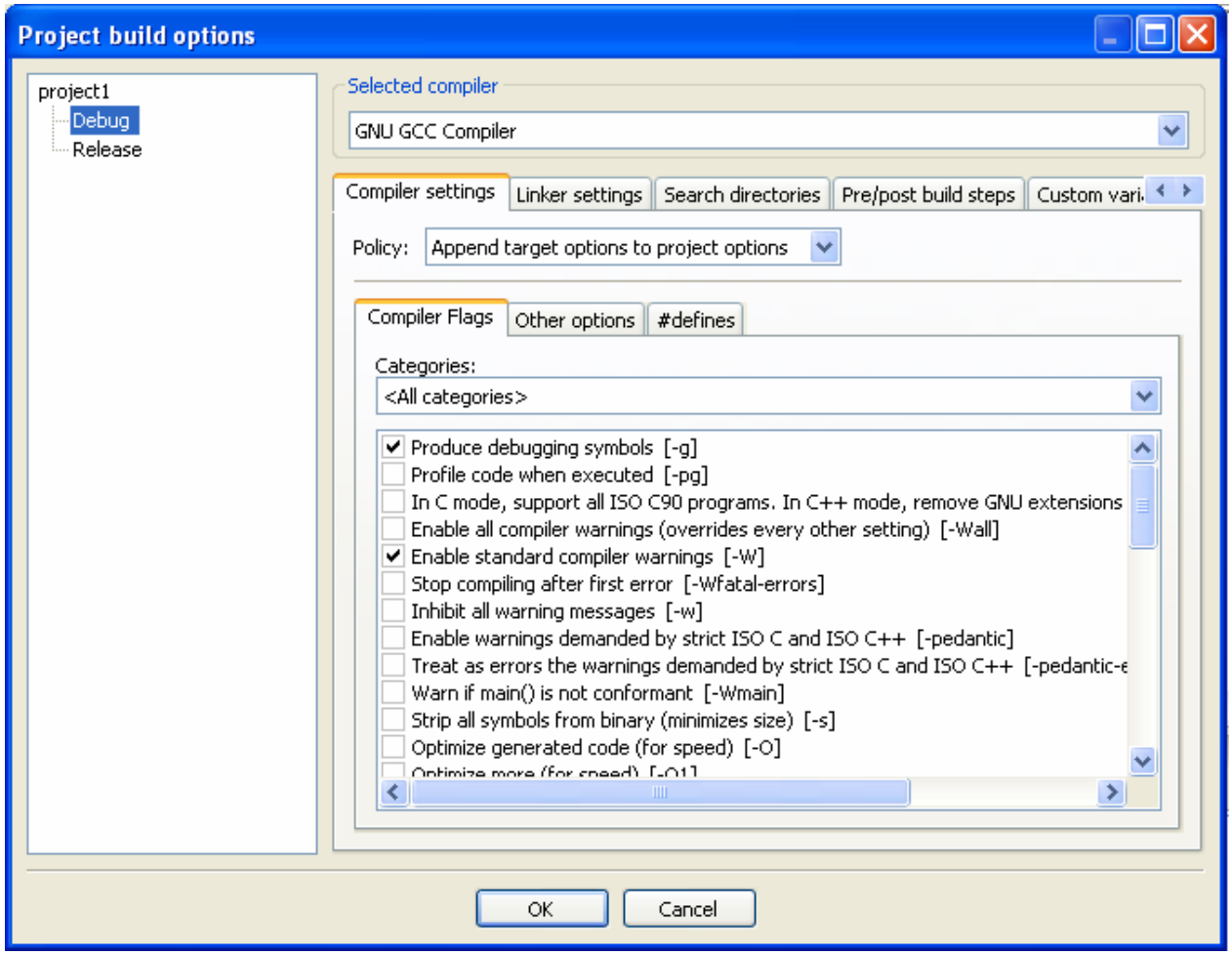




3.4 编译程序

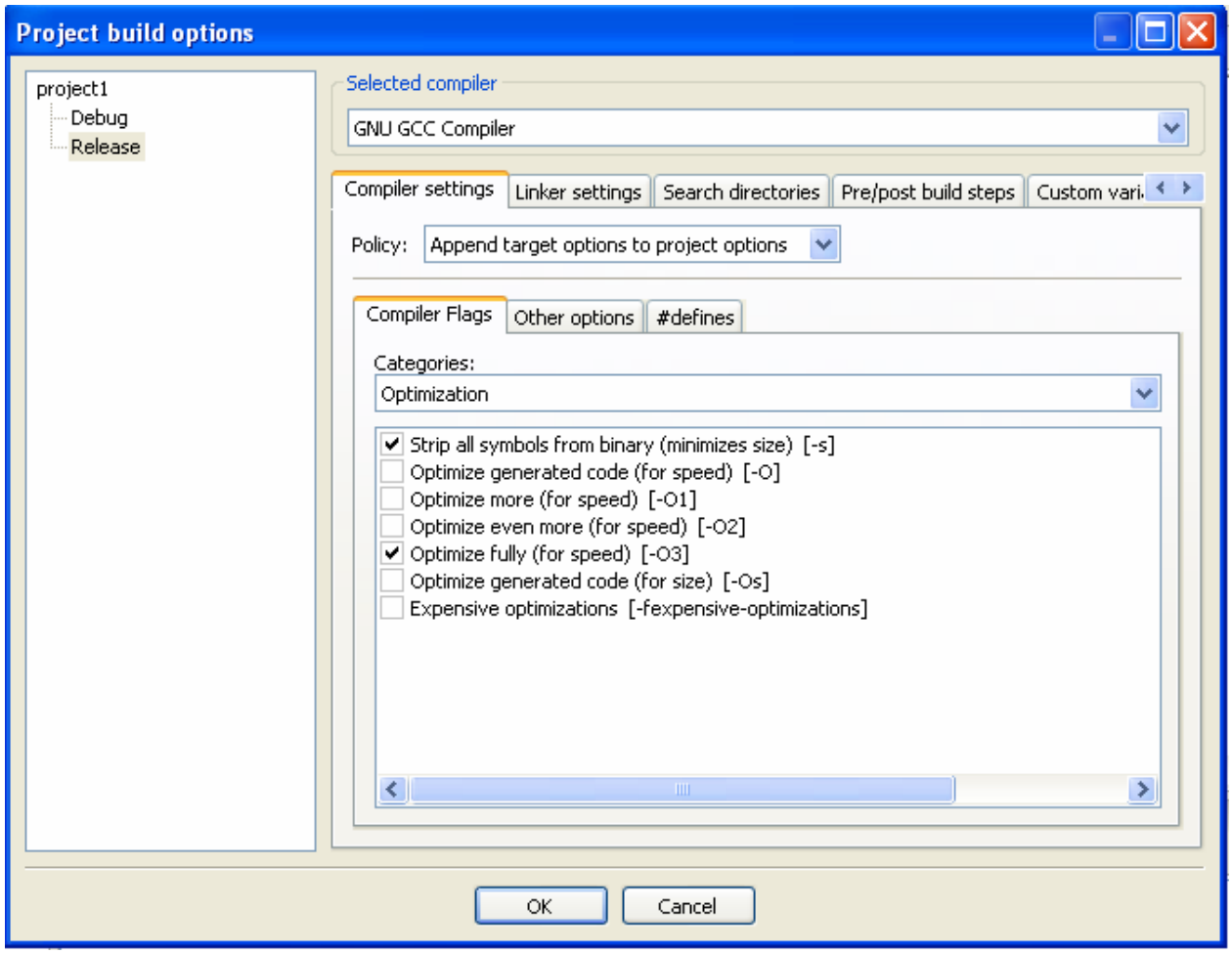
编译一个工程前先从Project的下拉菜单中找到Build options…选项，见右图。

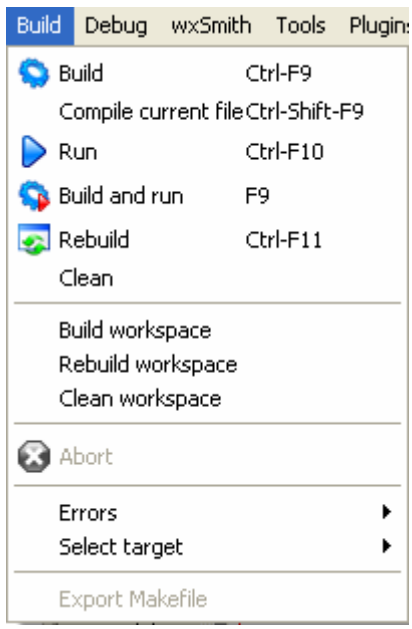


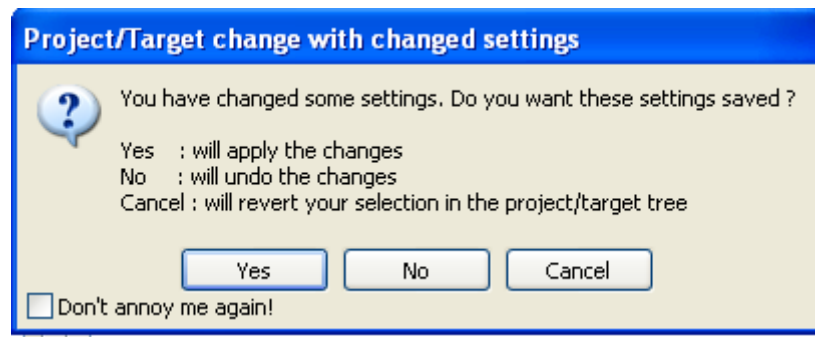


用鼠标点击Build options…按钮会弹出一个关于project1的对话框，有两个类别，debug和release，首先配置debug选项，一般而言，只关注Compiler Flags菜单下的两项，Producing debugging symbols [-g]和Enable standard compiler warnings [-W]。前者表示产生调试信息，后者意味着给出标准的编译警告信息，

分别打勾，见上图。然后配置Release，选择子菜单Compiler Flags下的Optimization选项，对于普通的应用，选择两项足矣，分别是Strip all symbols from binary (minimizes size) [-s]和Optimize fully (for speed) [-O3]，这两项前面打勾，见下图。







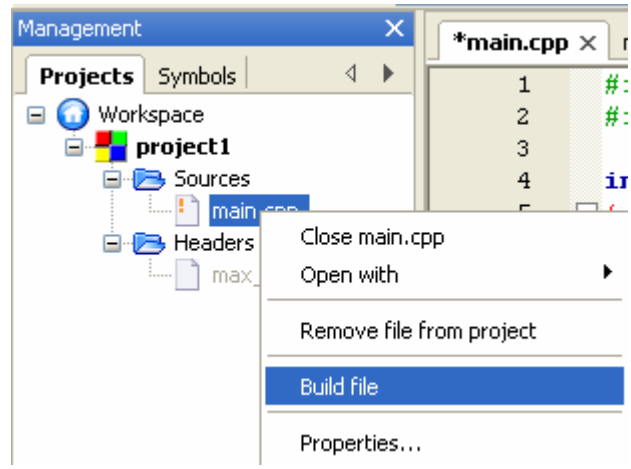
从debug切换到Release会弹出一个对话框，选择Yes。见上图。

编译一个工程或者文件的功能按钮都在Build下拉菜单，见左图。

此外，编译程序还有快捷菜单，见下图。

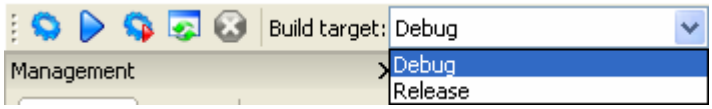


两个菜单中按钮图标相同的，功能也相同。表示编译当前工程，表示运行编译成功的文件，表示编译并运行。如果编译当前文件，可以使用 Build 下拉菜单中的按钮 Compile current file Ctrl-Shift-F9，也可以展开左侧的工程目录树，移动鼠标到需要单独编译的文件上面，利用鼠标右键弹出菜单中选择Build file按钮，见右图。

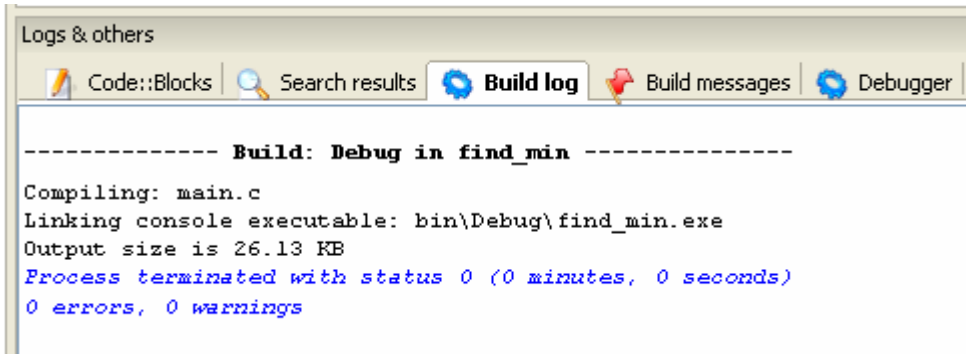


如果一次编译不成功，修改后重新编译可以使用按钮Rebuild，它代表重新编译，如果想清楚上次编译的目标文件，可以使用Build的下拉菜单中的Clean按钮。

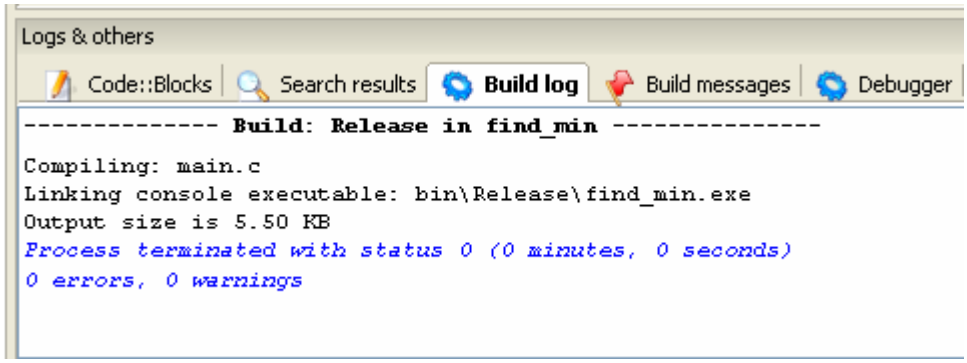
根据客观需要，我们可以把目标文件编译成debug或者release版本，用鼠标从快捷菜单上的Build target小窗口进行选择即可，见下图。



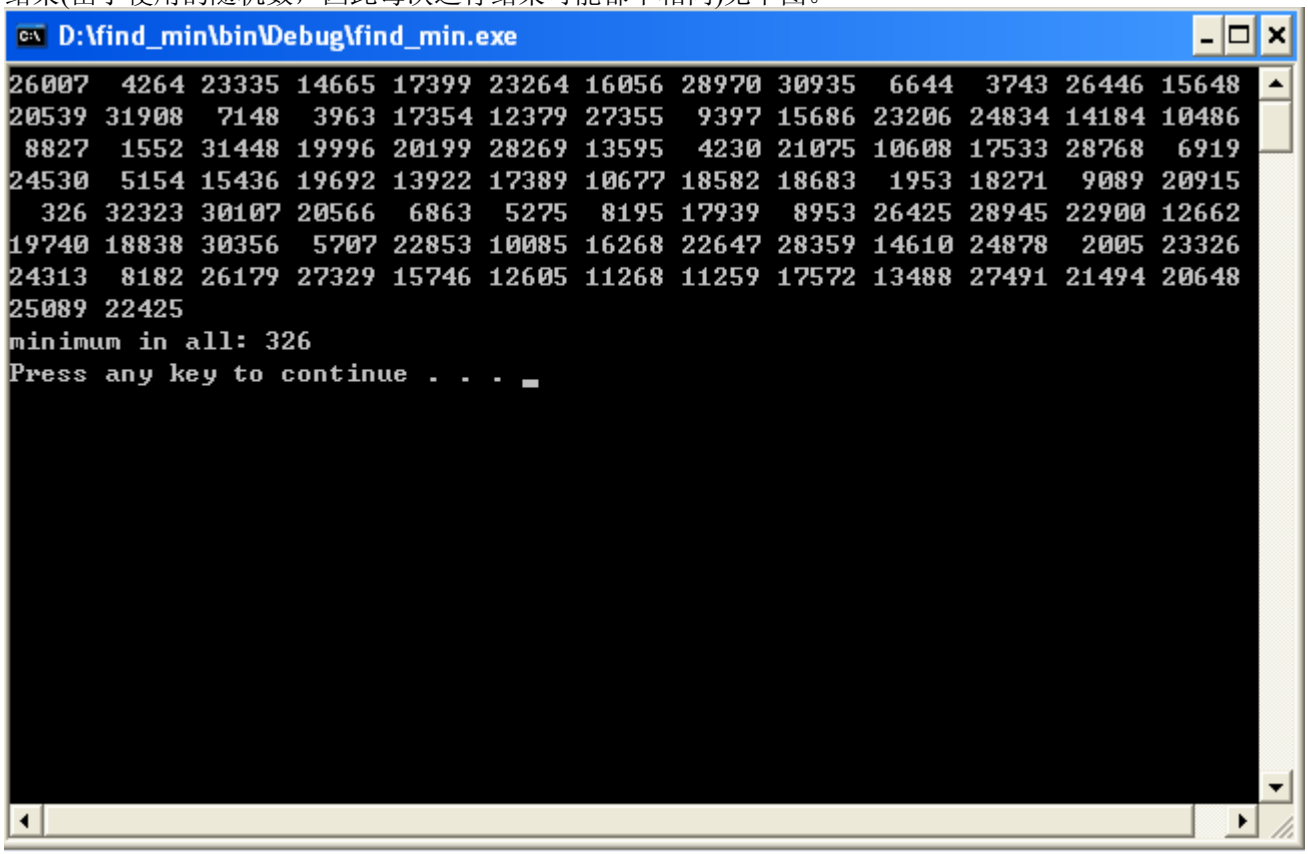
作为例子，我们用C语言编写一个求一个随机整数集合中最小值的程序。首先建立工程find_min，编写程序(程序源代码见本章的下一小节3.5.1)，然后编译。首先选择编译目标文件为debug版本，用鼠标点击快捷菜单上的按钮，则开始编译，编译完毕后日志窗口有一些提示信息，见下图。



我们选择目标文件为release版本再编译，编译完毕后日志窗口有一些提示信息，见下图。如下图。



运行该目标文件(debug或者release版本)，可以用鼠标点击快捷菜单上的按钮，则一种可能的运行结果(由于使用的随机数，因此每次运行结果可能都不相同)见下图。



生成debug和release版本的二进制文件运行结果相同，但它们二进制文件大小不同，此处find_min的debug版本二进制文件大小26.13KB，release版本的二进制文件大小5.5KB。由于前者包含了一些测试信息，所以它的二进制文件较大。

一次编译成功当然最好，但是很多时候不能一次编译成功，这时需要根据给出的出错信息修改源程序，然后重现编译，可能需要反复这个过程，直到编译成功。编译成功说明没有语法错误，但未必没有逻辑错误，程序中的逻辑错误需要您自己检查，当然了，您可以使用调试工具帮助检查逻辑错误。

再看一个例子，下面这个例子用来查看当前编译器的补白(padding)，也有很多人称补白为对齐(alignment)，补白是编译器为了提升程序执行速度，让不同数据类型占用同样的长度一起处理，从而减少CPU取指令次数，提升运算速度。下面给出的代码不能一次编译成功，需要我们找到问题所在，并改正其中的语法错误。

```
1   // padding.hpp
2
3   class padding
4   {
5   private:
6     char a;
7     int b;
8     double c;
9   public:
10    padding(char c = 'a', int i = 100, double d = 2.3) : a(c), b(i), c(d) {}
11  }
12
```

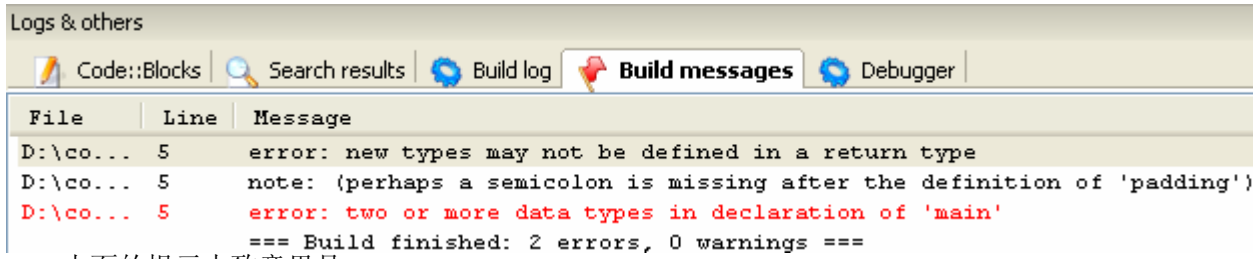
```cpp
// padding.cpp
#include <iostream>
#include "padding.hpp"

int main()
{
  padding a;
  std::cout << "sizeof(padding) == " << sizeof(padding) << '\n';
  std::cout << "(sizeof(char) + sizeof(int) + sizeof(double)) == "
            << sizeof(char) + sizeof(int) + sizeof(double) << '\n'

  char* p = (char*)(&a);
  int int_len = sizeof(int);
  std::cout << "1st element: " << *p << '\n';
  std::cout << "2nd element: " << *(int*)(p + int_len) << '\n';
  std::cout << "3rd element: " << *(double*)(p + (int_len << 1)) << '\n';

  return 0;
}
```

编译提示信息见如下贴图。



上面的提示大致意思是：

“出错：在返回类型后新类型必须定义”

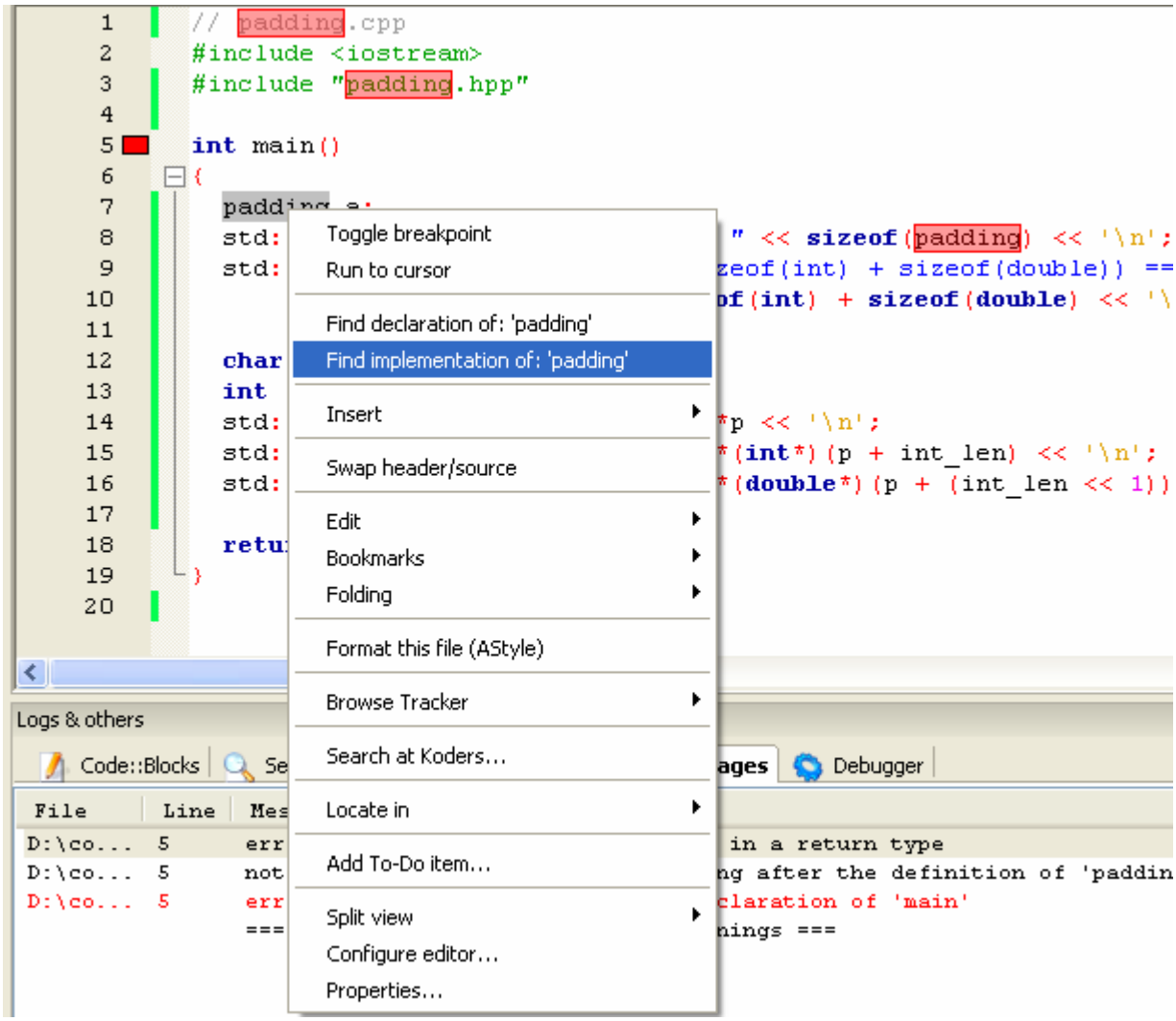
“注意：可能padding定义后少了一个分号”

“出错：main中声明了两个或者更多个类型”

修改源程序中的错误技巧性非常强，如果程序不能通过编译，那么根据出错的提示信息修改源程序时，能看懂哪些错误信息就先修改哪里的错误，有时可能源程序仅一两个错误，但编译器提示错误信息给出很多，这些错误信息中很大一部分可能没有任何帮助价值，我们需要找出对我们有意义的错误信息。有时我们发现，修改完一个错误重新编译后原来给出的很多错误信息突然变得少多了。

此外，如果您经常编程，最好能读懂编译器给出的提示信息。如果您的计算机专业英语非常好，那将对您的编程有很大作用，不仅仅是看编译器给出的提示信息方便了修改程序中的语法错误，而且编程过程中经常需要查阅函数的说明文档，计算机专业英语对于编程有很大帮助。

现在开始修改错误。我们首先去找padding的定义，看看是否忘记了一个分号。查找padding的声明或者实现很简单，选中padding，则padding变成了灰色，所有和padding相同的字符窜都变成了红色(注：笔者使用了比较新的Code::Blocks版本，如果您用的版本比较旧，则可能没有这样的功能，颜色可以设置，不一定是红色)，按下用鼠标右键在弹出的快捷菜单中选择按钮Find implementation of: 'padding'，见下图。

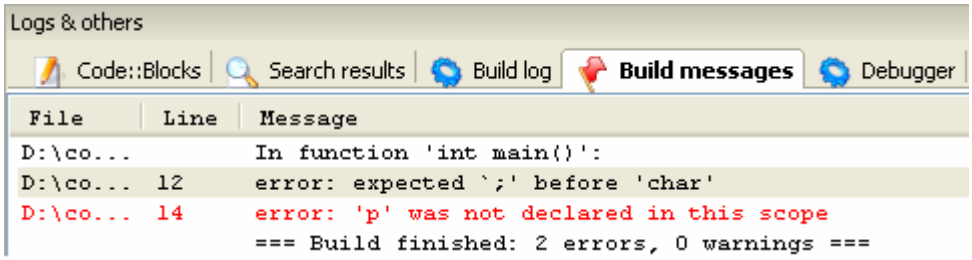


选中它后屏幕会自动切换到padding的实现，我们发现，padding类最后果然少了一个分号，见下图。

```
1  // padding.hpp
2
3  class padding
4  {
5  private:
6    char a;
7    int b;
8    double c;
9  public:
10   padding(char c = 'a', int i = 100, double d = 2.3) : a(c), b(i), c(d) {}
11 }
12
13
```

丢了一个分号

加上这个分号后点击按钮保存当前文件padding.hpp，然后点击按钮编译。结果又有出错信息，见下图。

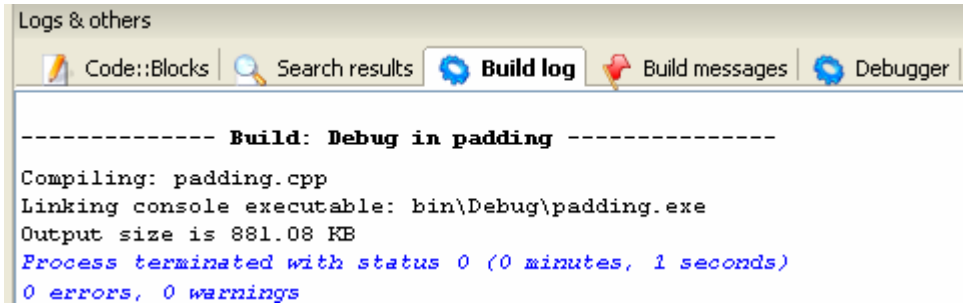


我们想修改哪行出错的代码，就用鼠标在下面的提示信息栏选中它，然后双击鼠标左键，则屏幕自动切换到那行处。

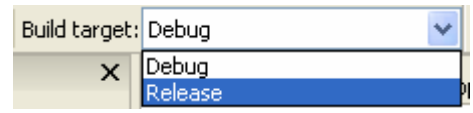
我们先不管第14行，看看第12行，提示说，char前本应该有;，第11行是个空行，不要忘记编译器会忽略掉空行和注释，因此我们查看前一行第10行，发现果然丢了一个分号。

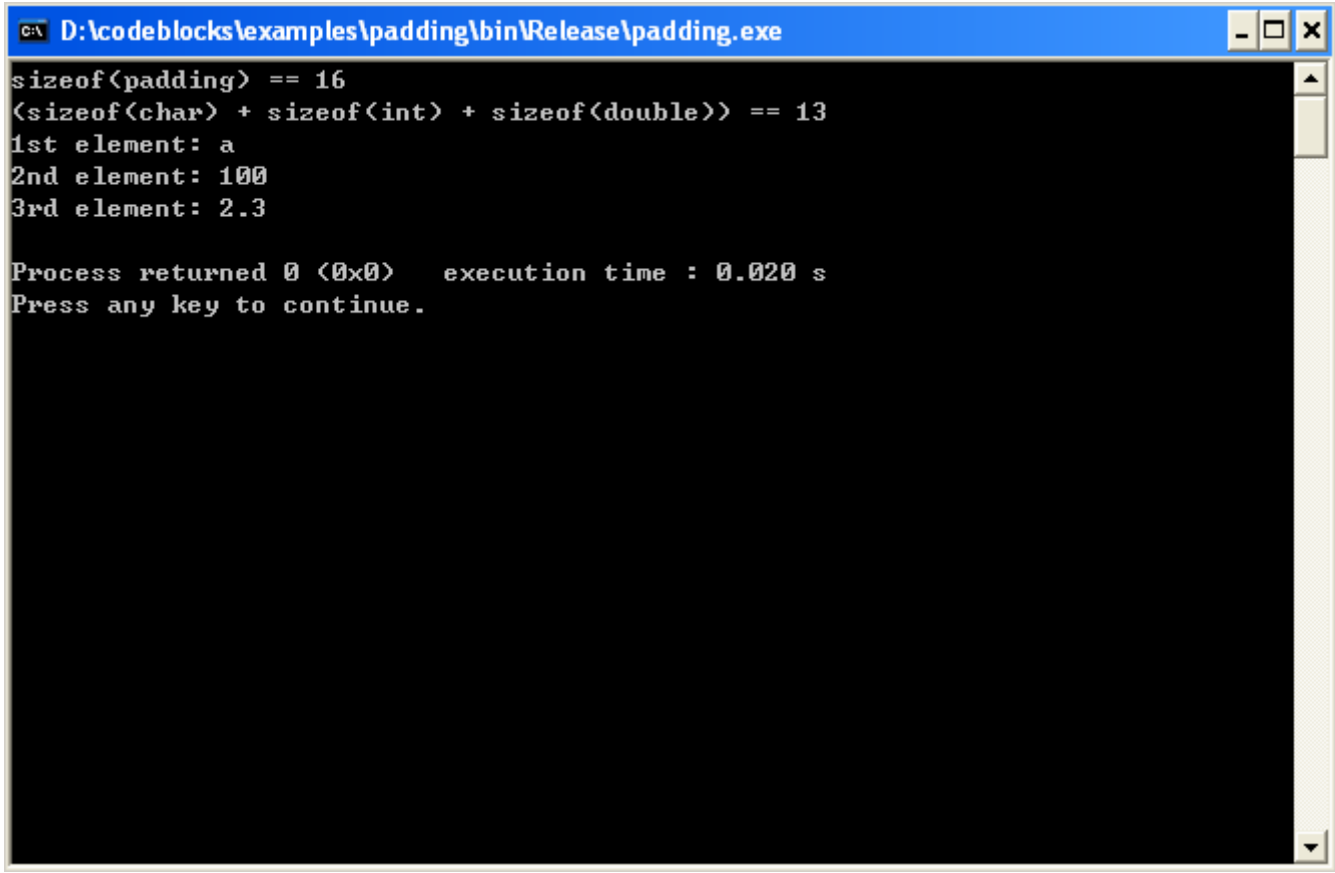
```
1   // padding.cpp
2   #include <iostream>
3   #include "padding.hpp"
4
5   int main()
6   {
7     padding a;
8     std::cout << "sizeof(padding) == " << sizeof(padding) << '\n';
9     std::cout << "(sizeof(char) + sizeof(int) + sizeof(double)) == "
10              << sizeof(char) + sizeof(int) + sizeof(double) << '\n'
11
12    char* p = (char*)(&a);                          丢了一个分号
13    int int_len = sizeof(int);
14    std::cout << "1st element: " << *p << '\n';
15    std::cout << "2nd element: " << *(int*)(p + int_len) << '\n';
16    std::cout << "3rd element: " << *(double*)(p + (int_len << 1)) << '\n';
17
18    return 0;
19  }
20
```

补上一个分号，保存，然后继续编译。这下编译成功了，提示信息见下图。



再编译一个release版本(Build target:栏目选择Release然后编译)运行之，结果见下图。





注意：编译以上程序需要事先建立工程padding并设置好相应编译选项(Build options…)，跟前面建立Project1时设置编译选项类似，为了节约篇幅这里就没有赘述。因为系统默认一些选项，所以大多数情况下不用理会编译选项的设置也能正常编译程序，但是编译程序时产生的提示信息未必就是您感兴趣和期望的，例如，有人可能希望产生标准的出错和警告信息，有人则可能希望产生任何可能的出错和警告信息，个别人则希望什么提示信息都没有从而省得心烦，再者，期望产生目标代码的类型可能也不同，有人可能希望最高级优化，有人则可能希望专门针对某种CPU优化，还有人可能认为是否优化无所谓。鉴于此，最好根据个人喜好和实际情况设置这些编译选项。

我们接下来再看一个例子。以下程序用来交换两个不同的值，由于我们并不知道要交换的值的类型，所以就使用模板，参数类型用指针，当交换的两个值是整数的时候进行特化，参数类型用引用。为了知道调用了那个swap函数进行交换，在swap函数体里面我们附加输出一点信息。

跟往常一样，我们先建立一个工程，工程名取undefined，然后配置该工程的调试和编译参数。配置这些选项：☑ Enable all compiler warnings (overrides many other settings) [-Wall]以便让编译器给出标准的提示信息，当生成debug版本的二进制文件时，☑ Produce debugging symbols [-g]，产生调试信息，当需要生成release版本的二进制文件时，☑ Optimize fully (for speed) [-O3]，进行三级优化，让生成的二进制文件占用空间比较小，☑ Strip all symbols from binary (minimizes size) [-s]。建议读者自行配置以上几个编译参数选项作为练习。

```cpp
// swap.hpp
#include <iostream>

template <class T>
void swap(T* x, T* y)
{
  std::cout << "swap(T*, T*) called.\n";
  if(*x != *y)
  {
    T tmp = *x;
    *x = *y;
    *y = tmp;
  }
}

void swap(int& x, int& y)
{
  std::cout << "swap(int&, int&) called.\n";
  if(x != y)
  {
    x ^= y;
    y ^= x;
    x ^= y;
  }
}

```

```cpp
// main.cpp
#include <iostream>

using namespace std;

#include "swap.hpp"

int main()
{
  int a, b = 1;
  swap(a, b);
  cout << a << ' ' << b << '\n\n';

  double c = 3.3, d;
  swap(c, d);
  cout << c << ' ' << d << '\n\n';

  char e = 'e', f = 'f';
  swap(&e, &f);
  cout << e << ' ' << f << '\n\n';

  char* str1 = "string 1", str2 = "string 2";
  swap(str1, str2);
  cout << str1 << ' ' << str2 << '\n\n';

  return 0;
}

```

编译以上程序，产生如下提示信息。

```
22    char* str1 = "string 1", str2 = "string 2";
```

Logs & others

Code::Blocks | Search results | Build log | Build messages | Debugger

File	Line	Message
D:\codeblocks\...	12	warning: multi-character character constant
D:\codeblocks\...	16	warning: multi-character character constant
D:\codeblocks\...	20	warning: multi-character character constant
D:\codeblocks\...	24	warning: multi-character character constant
D:\codeblocks\...		In function 'int main()':
D:\codeblocks\...	22	warning: deprecated conversion from string constant to 'char*'
D:\codeblocks\...	22	error: invalid conversion from 'const char*' to 'char'
D:\codeblocks\...	23	error: invalid initialization of reference of type 'int&' from expression of type 'char*'
D:\codeblocks\...	16	error: in passing argument 1 of 'void swap(int&, int&)'
		=== Build finished: 3 errors, 5 warnings ===

如此简单的一个小程序，居然产生了5个警告信息，而且编译器还报告3个错误。

首先我们看警告信息，警告说第12, 16, 20, 24行多个字符字符常量，第22行，不建议把字符窜常量转换成char*。用鼠标双击第一行编译信息，则自动跳到了main函数的第12行，前面出现一个红色的方框，见下图。

```
3
4   using namespace std;
5
6   #include "swap.hpp"
7
8   int main()
9   {
10    int a, b = 1;
11    swap(a, b);
12    cout << a << ' ' << b << '\n\n';
13
14    double c = 3.3, d;
15    swap(c, d);
16    cout << c << ' ' << d << '\n\n';
17
18    char e = 'e', f = 'f';
19    swap(&e, &f);
20    cout << e << ' ' << f << '\n\n';
21
```

Logs & others

Code::Blocks | Search results | Build log | Build messages | Debugger

File	Line	Message
D:\codeblocks\...	12	warning: multi-character character constant
D:\codeblocks\...	16	warning: multi-character character constant
D:\codeblocks\...	20	warning: multi-character character constant
D:\codeblocks\...	24	warning: multi-character character constant
D:\codeblocks\...		In function 'int main()':
D:\codeblocks\...	22	warning: deprecated conversion from string constant to 'char*'
D:\codeblocks\...	22	error: invalid conversion from 'const char*' to 'char'
D:\codeblocks\...	23	error: invalid initialization of reference of type 'int&' from expression of type 'char*'
D:\codeblocks\...	16	error: in passing argument 1 of 'void swap(int&, int&)'
		=== Build finished: 3 errors, 5 warnings ===

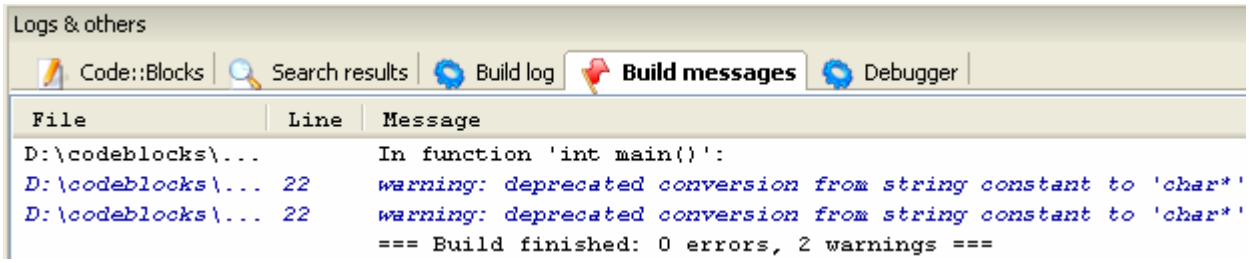
果然，把两个换行符冠以单引号了，两个字符组成了一个字符窜，应该冠以双引号，再看其它行上的几个，出现同样的问题了。用Replace一次性全部取代改正过来并保存当前文件。再看第22行，我们使用的是C语言风格定义了字符窜，语法上没有问题，暂且不用管它。

接下来，看看编译错误信息。第22行，把const char*类型转换成char类型。仔细看一看发现第二个

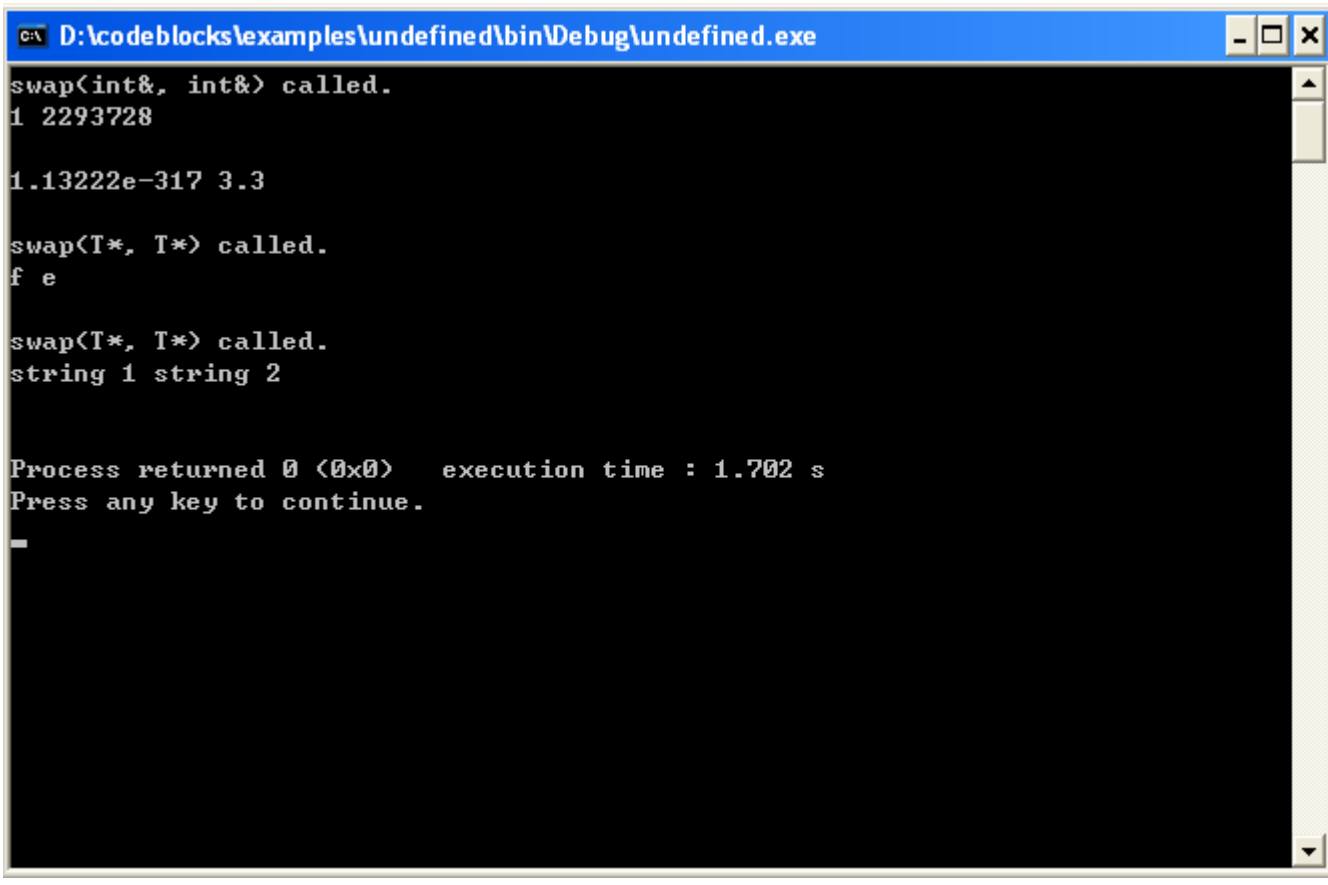
字符窜定义少些了一个*，结果编译器认为是把”string 2”这个常量字符窜赋值给一个字符变量str2了，见下图。

```
21
22      char* str1 = "string 1", str2 = "string 2";
23      swap(str1, str2);
24      cout << str1 << ' ' << str2 << "\n\n";
25
26      return 0;                    漏掉了一个*
27  }
```

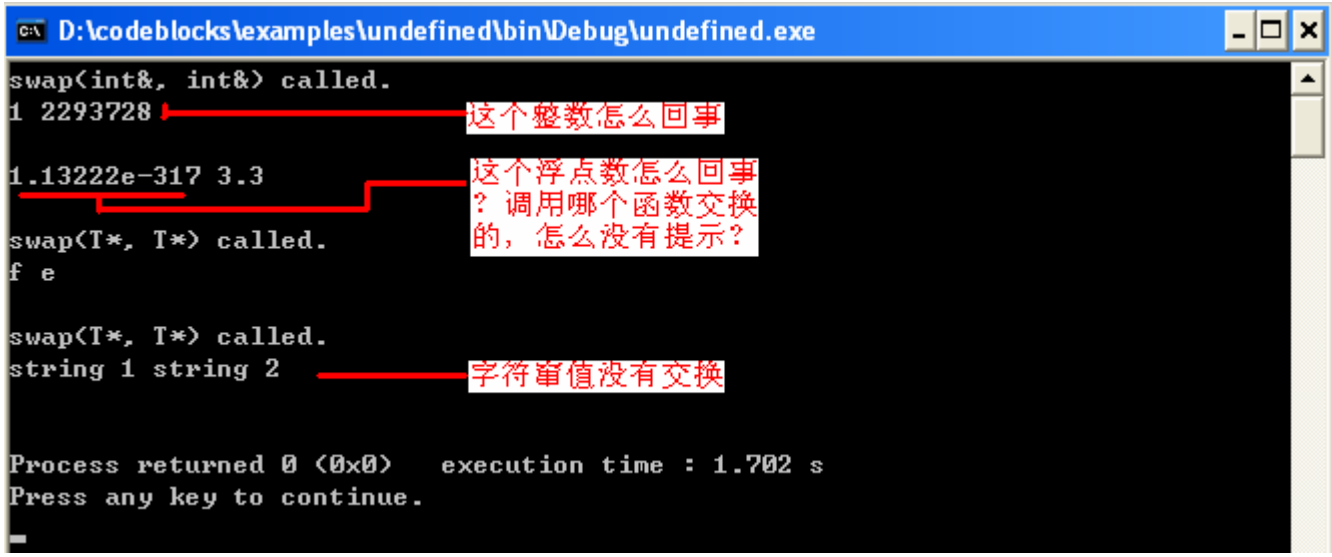
加上*，保存当前文件，然后再编译，这次的编译提示信息如下。



只有警告不建议的C风格字符窜常量转换了。暂且不用理会，先运行看一下结果再说。



结果并不理想，疑点很多，见下图中红色文字标示。




```
 8  int main()
 9  {
10      int a, b = 1;
11      swap(a, b);
12      cout << a << ' ' << b << "\n\n";
13
14      double c = 3.3, d;
15      swap(c, d);
16      cout << c << ' ' << d << "\n\n";
17
18      char e = 'e', f = 'f';
19      swap(&e, &f);
20      cout << e << ' ' << f << "\n\n";
21
22      char* str1 = "string 1", * str2 = "string 2";
23      swap(str1, str2);
24      cout << str1 << ' ' << str2 << "\n\n";
```

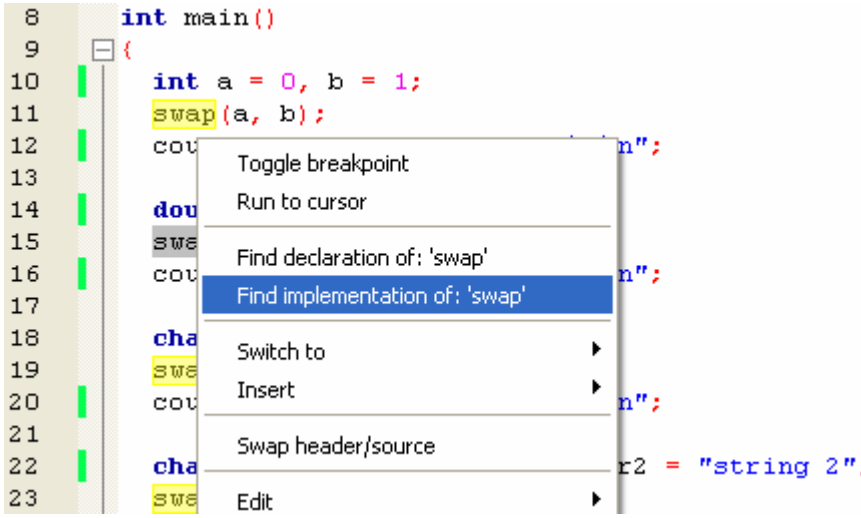
局部变量a没有赋值

局部变量d没有赋值

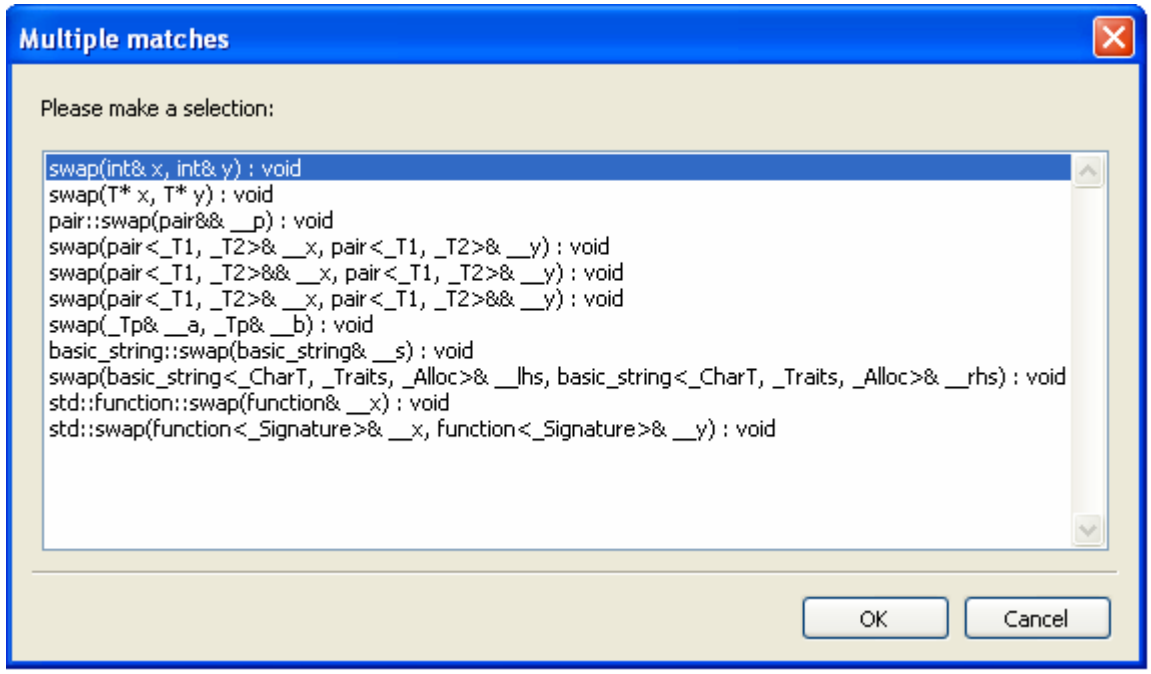
调用哪个swap?

为什么不能交换两个字符串的值

先改正容易修改的错误，给a和d赋初值，假设a = 0, d = 0.1，保存当前文件。然后看看第15行，到底调用了哪个swap，因为我们的swap.hpp没有跟这个调用相匹配的函数，去能编译不出现语法错误，实在蹊跷，那我们就看看到底调用了哪个swap。选中swap(显示灰色)，按下鼠标右键，在弹出的快捷菜单中选择Find implementation of: 'swap'，见下图。



弹出一个对话框，显示了所有可能的swap函数头，见下图。



居然有这么多的swap，前两个是我们自己定义的，后面的是系统自带的。不要忘记前面的using namespace std;这会污染整个std命名空间，这样使用是OOP的大忌！注释掉using namespace std;，加一行using std::cout;然后重新编译，看看结果如何。

```
1   // main.cpp
2   #include <iostream>
3
4   //using namespace std;
5   using std::cout;
6   #include "swap.hpp"
7
8   int main()
9   {
10    int a = 0, b = 1;
11    swap(a, b);
12    cout << a << ' ' << b << "\n\n";
13
14    double c = 3.3, d = 0.1;
15    swap(c, d);
16    cout << c << ' ' << d << "\n\n";
17
18    char e = 'e', f = 'f';
19    swap(&e, &f);
20    cout << e << ' ' << f << "\n\n";
```

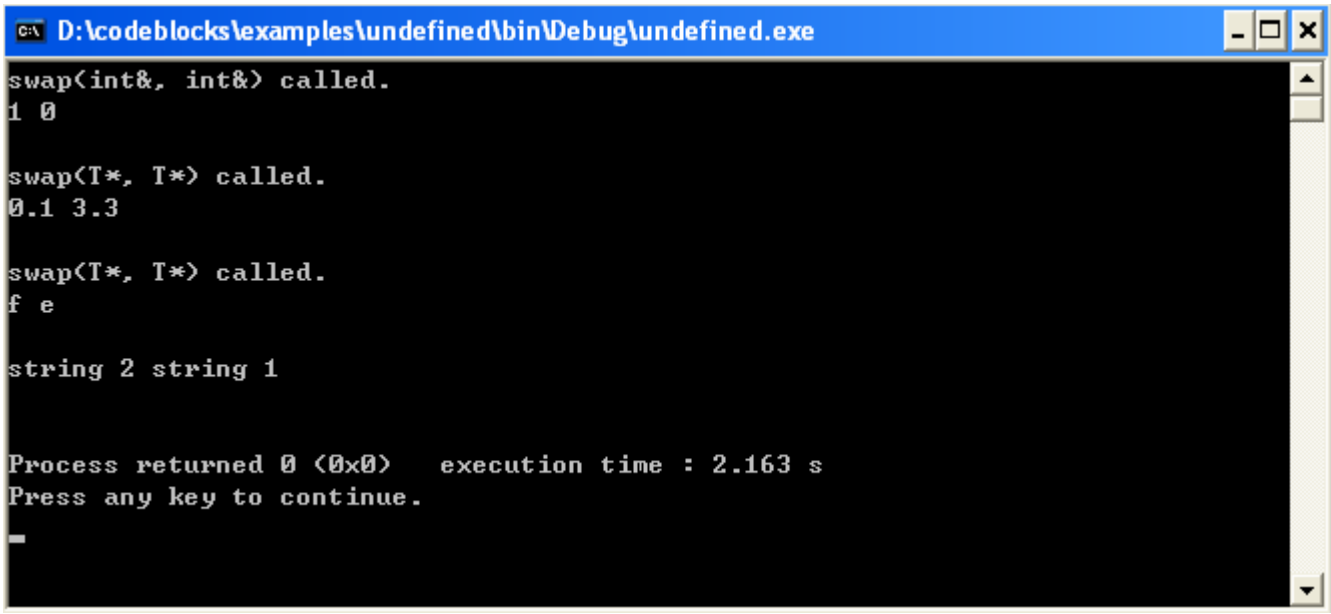
Logs & others

Code::Blocks | Search results | Build log | **Build messages** | Debugger | Thread search

File	Line	Message
D:\codeblocks\...		In function 'int main()':
D:\codeblocks\...	15	error: invalid initialization of reference of type 'int&' from
D:\codeblocks\...	16	error: in passing argument 1 of 'void swap(int&, int&)'
D:\codeblocks\...	22	warning: deprecated conversion from string constant to 'char*'
D:\codeblocks\...	22	warning: deprecated conversion from string constant to 'char*'
		=== Build finished: 2 errors, 2 warnings ===

最可能的匹配是void swap(int&, int&)，但是参数类型却不合适，因此编译器报错。知道原因就好，改正方法非常简单，改成swap(&c, &d);就可以了。

接下来，我们看看为何两个字符窜的值不能交换吧。从输出信息可知，调用void swap(T*, T*);但是这个函数实现中有个判断是否相等的比较if(*x != *y)，而此刻的*x和*y都是字符's'(两个字符窜的首个字符相等)，自然后面的三个语句根本就不执行，不要忘记判断字符窜是否相等的C语言库函数是strcmp，而绝非直接判断是否相等。那怎么修改呢？最简单的办法是用C++的string处理这个问题，把字符窜当成对象来处理。main函数前加一行#include <string>，然后用std::string定义字符窜，即std::string str1("string 1"), str2("string2");，交换两个字符窜最好的办法是用std::string的成员函数swap，这样不需要复制字符窜内容，仅仅交换指针就可以了，也就是str1.swap(str2);。讲过上述这些修改后，保存，重新编译并运行得到的结果如下图。



3.5 调试程序

伴随着编写的程序愈来愈复杂，我们往往很难一次性编译成功并运行得到我们期望的结果(虽无语法错误，但可能有逻辑错误)，这时需要对程序进行调试以便定位错误。调试程序有时需要在程序中的某些地方设置一些特殊"点"，让程序运行到该位置停下来，有时需要检查某些变量的值，以帮助我们检查程序中的逻辑错误。

调试程序之前，首先要确认已经设置了Produce debugging symbols [-g]选项，按照如下顺序去找(-> 表示下一步)：Project -> Build options… -> find_min -> Debug -> Compiler Flags -> Produce debugging symbols [-g]，如果该项已经配置，则可以调试程序了，否则前面打勾，点击OK按钮设置好。

下面将会通过具体的例子简要讲述调试器的使用。

3.5.1 面向过程风格的程序

假设我们有一个随机数集合，该集合内随机数个数随机(为了便于观察结果是否正确，我们假设该集合中随机数少于100个)，要求找出最小的那个随机数。

从一批数据中找到我们需要的某一个方法很多，为了简单起见，我们使用顺序搜索算法。用C语言编写程序实现以上题目要求，写了三个文件，一个源文件(main.c)，两个头文件，其中一个头文件(find_min.h)用来找出最小的随机数，一个头文件(output_all.h)用来输出这个随机数集合中的所有随机数，代码见下贴图。

```c
// main.c
#include <stdio.h>    // for printf
#include <stdlib.h>   // for malloc, free, srand, rand, system
#include <time.h>     // for time
#include "find_min.h"
#include "output_all.h"

int main()
{
  srand(time(NULL)); // 用时间作为随机数种子
  // 随机数为SZ个, 0 <= SZ < 100
  const unsigned MOD = 100, SZ = rand() % MOD;
  // 在堆上开辟一段空间, 存储产生的随机数
  unsigned* arr = (unsigned*)malloc(SZ * sizeof(SZ));
  unsigned k = 0;
  for(; k < SZ; ++k)
    arr[k] = rand();

  output_all(arr, SZ);  // 输出产生的所有随机数
  // 找到最小的那个随机数, 并返回它的下标
  unsigned min = find_min(arr, SZ);
  // 如果集合非空必然存在最小值, 输出它
  if(0 != SZ)
    printf("minimum in all: %d\n", arr[min]);

  free(arr);    // 释放分配的堆内存空间

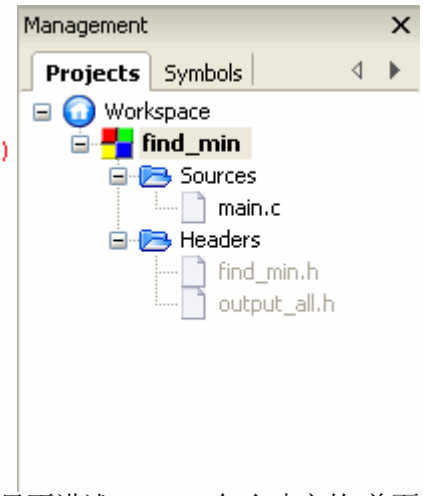
  system("PAUSE");  // 让系统等待, 以便我们查看结果是否正确
  return 0;
}
```

```c
// output_all.h
#ifndef OUTPUT_ALL_H_
#define OUTPUT_ALL_H_

#include <stdio.h>  // for printf
void output_all(const unsigned* v, unsigned n)
{
  // 输出结果中每行最多12个随机数
  const unsigned NL = 12, M = NL + 1;
  unsigned i = 0;
  for(; i < n; ++i)
  {
    printf("%5d ", v[i]);
    if(NL == i % M)  // 控制换行
      printf("\n");
  }
  printf(0 == n ? "" : "\n");  // 随机数集合为空不换行
}

#endif
```

```
1   // find_min.h
2   #ifndef FIND_MIN_H_
3   #define FIND_MIN_H_
4
5   unsigned find_min(const unsigned* v, unsigned n)
6   {
7     unsigned j = 0, i = 1;
8     for(; i < n; ++i)
9       if(v[i] < v[j])
10        j = i;  // 记住最小随机数的下标
11    return j;
12  }
13
14  #endif
15
```



上面贴图中，右图是工程find_min的展开图。为了节约篇幅，这里不讲述find_min怎么建立的(前面已经讲述了怎么建立工程)，也不详细解释代码(代码中有注释)，仅仅讲调试过程。

首先，要在程序中设置一个点，让程序运行到这个位置暂停，把光标置于您想启动跟踪程序运行过程的这行代码的前一行，再用Debug下拉菜单中或者调试程序快捷菜单上的Run to Cursor按钮启动调试器。(启动调试器的另一种方法是，用Debug下拉菜单中的Toggle breakpoint按钮先设置断点，然后点击Start开始按钮来启动调试器。)

接下来就要告诉跟踪程序运行到哪里再次停下来，以便我们检查此刻的运行结果，我们可以把光标置于该行代码前，然后再次点击Run to Cursor按钮，则程序运行到此处就停下来，此处称谓断点，可以用Debug下拉菜单中的Toggle breakpoint按钮设置断点，在一个断点前再次Toggle breakpoint，则该段点就被删除，如果想删除所有断点，可以用Debug下拉菜单中的Remove all breakpoints按钮。

如果我们希望每次总是运行到程序下一行前面，则可以点击Debug下拉菜单中或者调试程序的快捷菜单上按钮Next line，如果我们不想每次执行一行代码，希望执行的单位更小，可以逐条执行指令（Next instruction），如果运行到某个程序块(例如，调用某个函数)还可以选择按钮Step into，则运行到该代码块内，如果希望跳出该代码块，则可以选择按钮Step out。如果希望终止调试器，可以选择Continue按钮，则调试器会自然运行到结束，也可以用Stop debugger，则调试器就会被强行终止，调试过程结束。

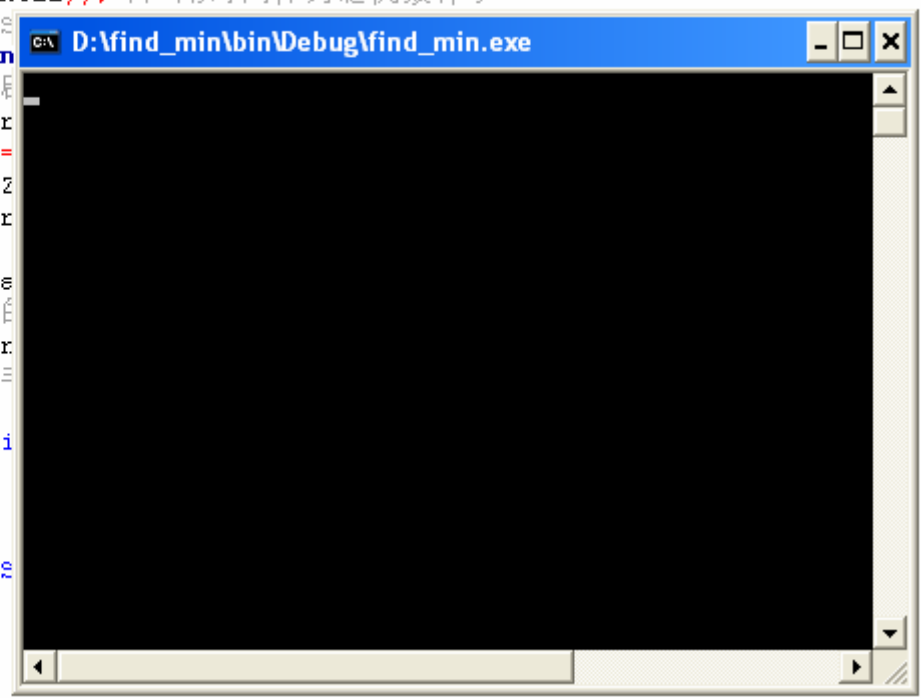
在程序执行到输出语句前，屏幕始终不显示任何信息。如果我们希望查看程序中变量的值，则可以从Debug下拉菜单中选择Debugging windows子菜单的Watches按钮，也可以从快捷菜单按钮下找到Watches按钮。

下面用示意图说明调试find_min的过程。

(1) 启动调试器

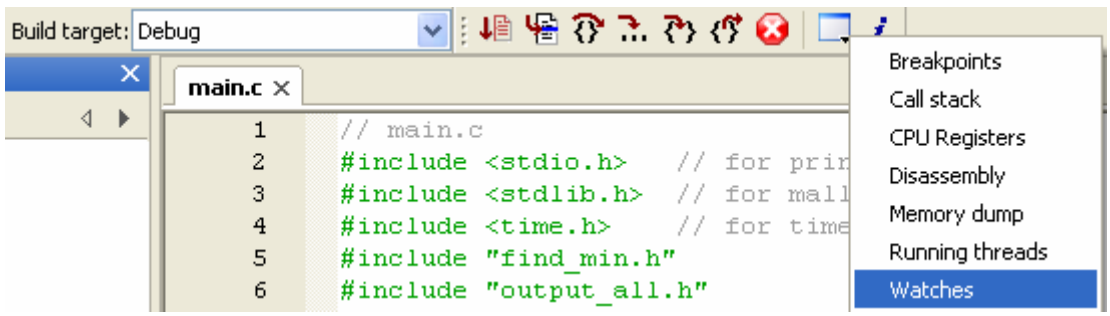
把光标置于int main之前，点击Run to cursor按钮，则可以看到{前面有个黄色的箭头，而且还会出现了一个没有任何输出信息的对话框，如下图。

```
 8   int main()
 9   {
10     srand(time(NULL)); // 用时间作为随机数种子
11     // 随机数为s
12     const unsign
13     // 在堆上开b
14     unsigned* ar
15     unsigned k =
16     for(; k < SZ
17       arr[k] = r
18
19     output_all(a
20     // 找到最小的
21     unsigned min
22     // 如果集合s
23     if(0 != SZ)
24       printf("mi
25
26     free(arr);
27
28     system("PAUS
29     return 0;
30   }
31
```



(2) 打开观察窗口

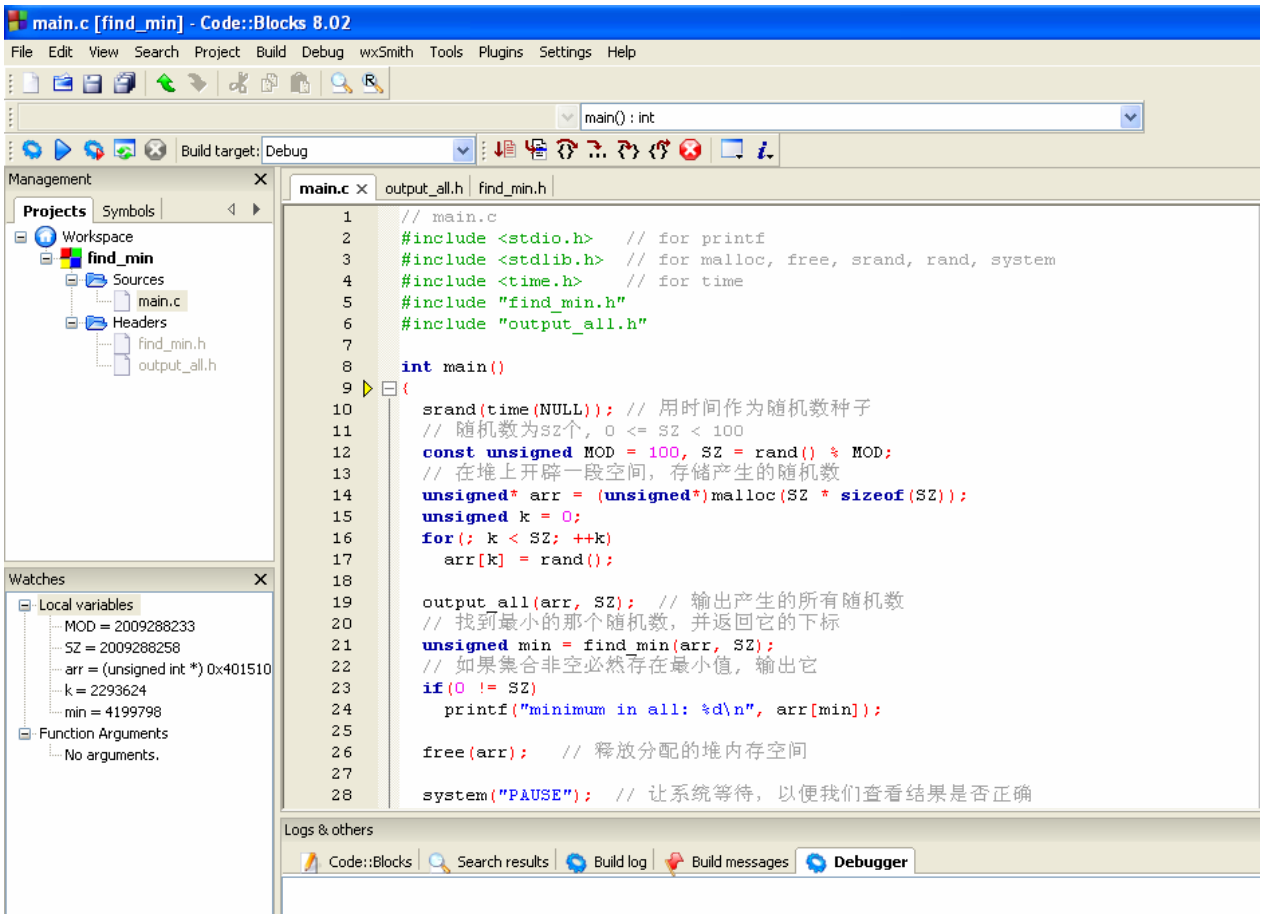
为了查看程序运行中变量值的变化情况，需要打开观察变量的窗口，用快捷菜单上按钮的下拉菜单中Watches打开即可(也可以用Debug -> Debugging windows -> Watches打开)。



Watches窗口见右图。



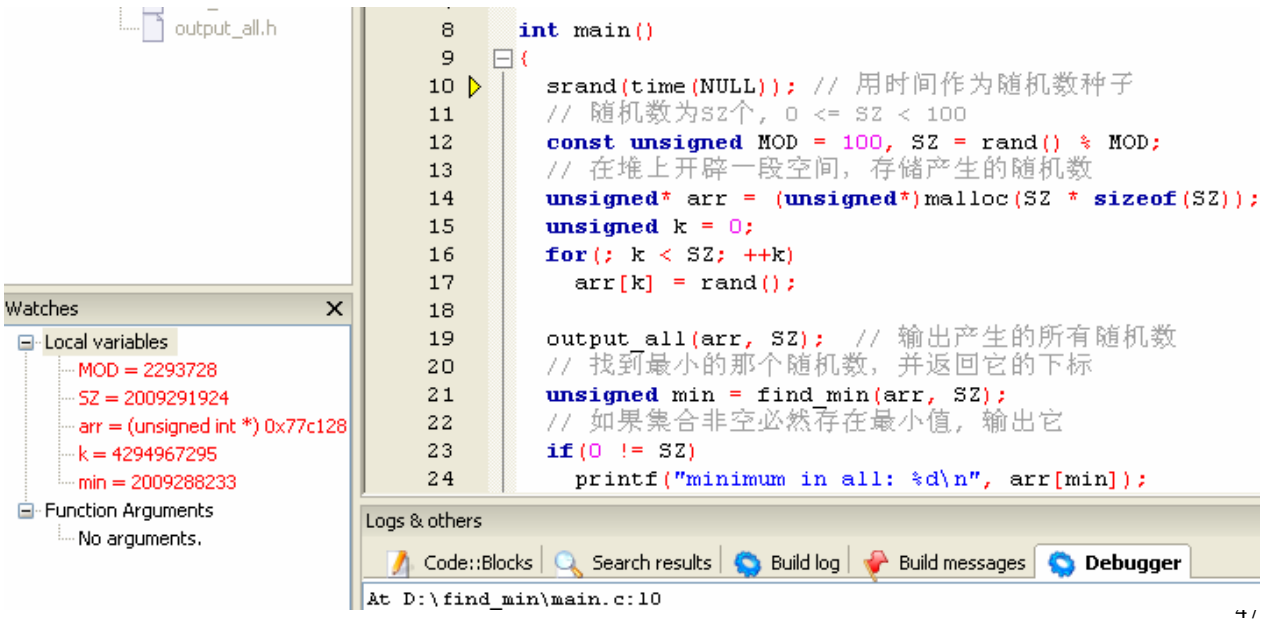
为了方便观察整个调试过程布局，拖动Watches窗口到左下角，并展开各个变量，如下图。



此时，从Watches窗口中可以看到定义的局部变量都是随机值，因为程序尚未执行到给这些变量赋值的语句。

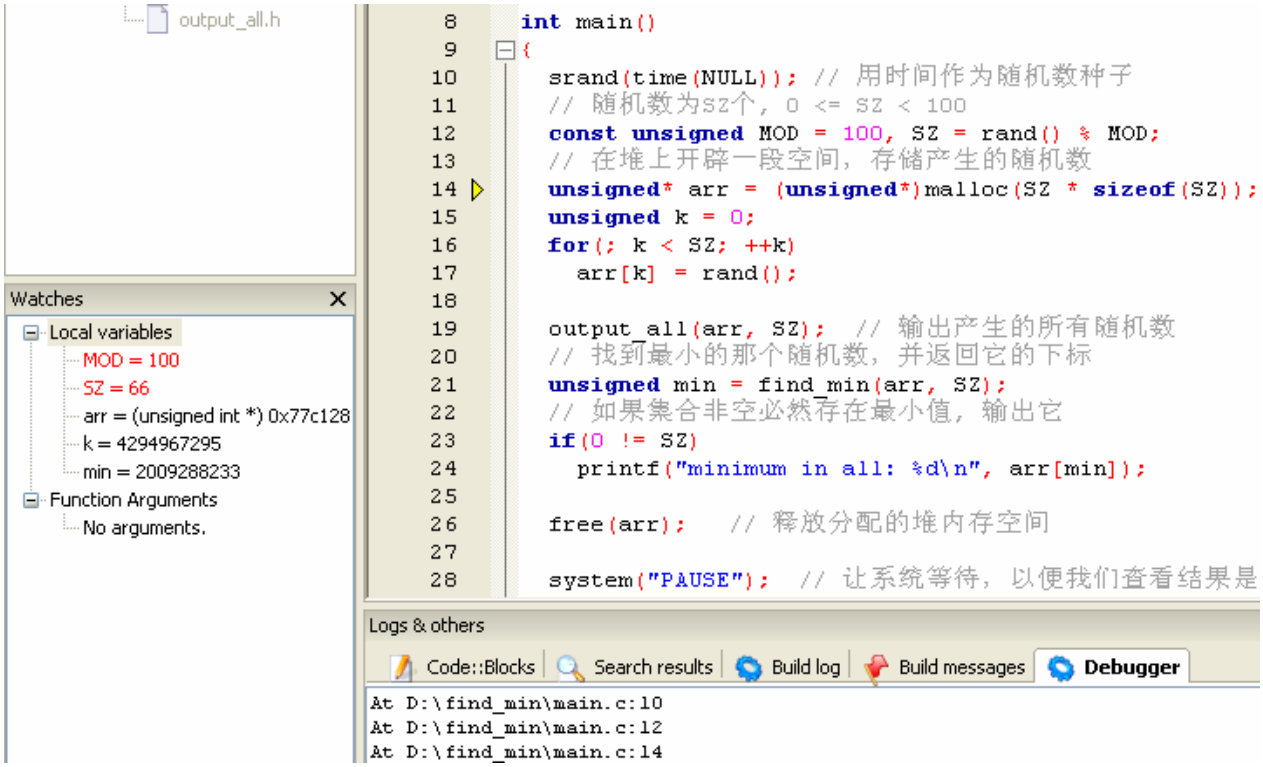
(3) 执行到下一行(第10行代码)

见下面的截图，输出信息窗口仍然什么都不显示，观察变量值的窗口局部变量的值都是系统赋予的随机值(左下角红色文字部分)。



(4) 执行到第14行

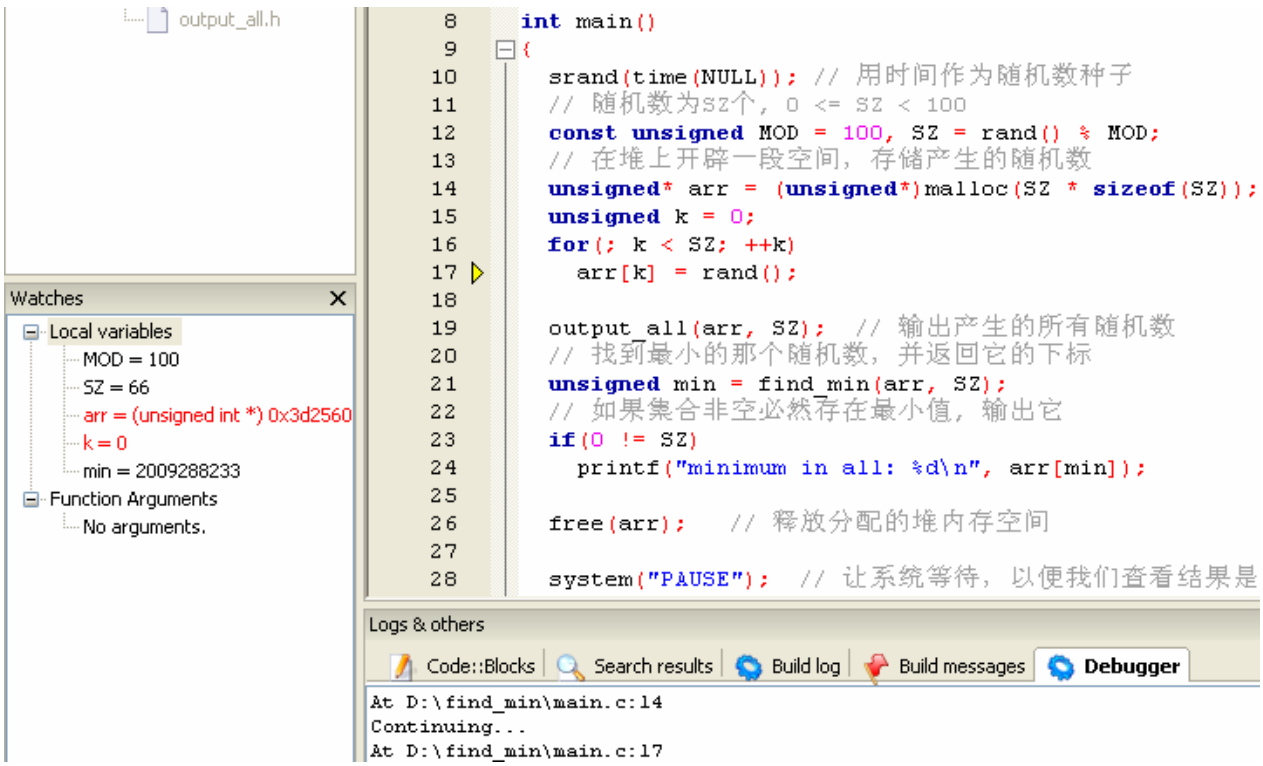
把光标放到第14行代码前面，点击Run to Cursor按钮，见下图。



由于第12行语句已经被执行，从观察窗口可以看到局部变量MOD = 100, SZ = 33 (观察窗口红色文字部分)，此外日志窗口的调试器Debugger栏目也出现了一些文字，显示了main.c被执行的代码。

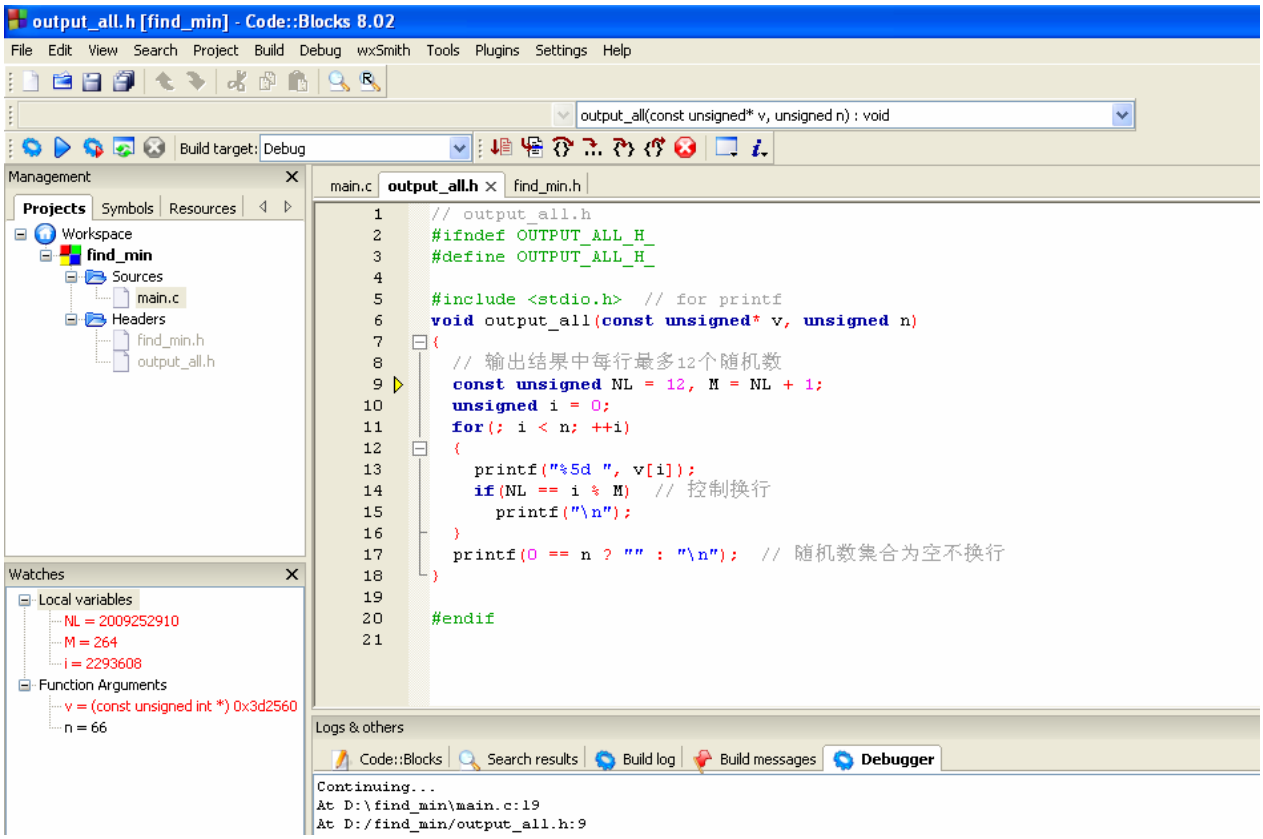
(5) 执行到第17行

把光标放到第17行代码前面，点击Run to Cursor按钮，见下图。

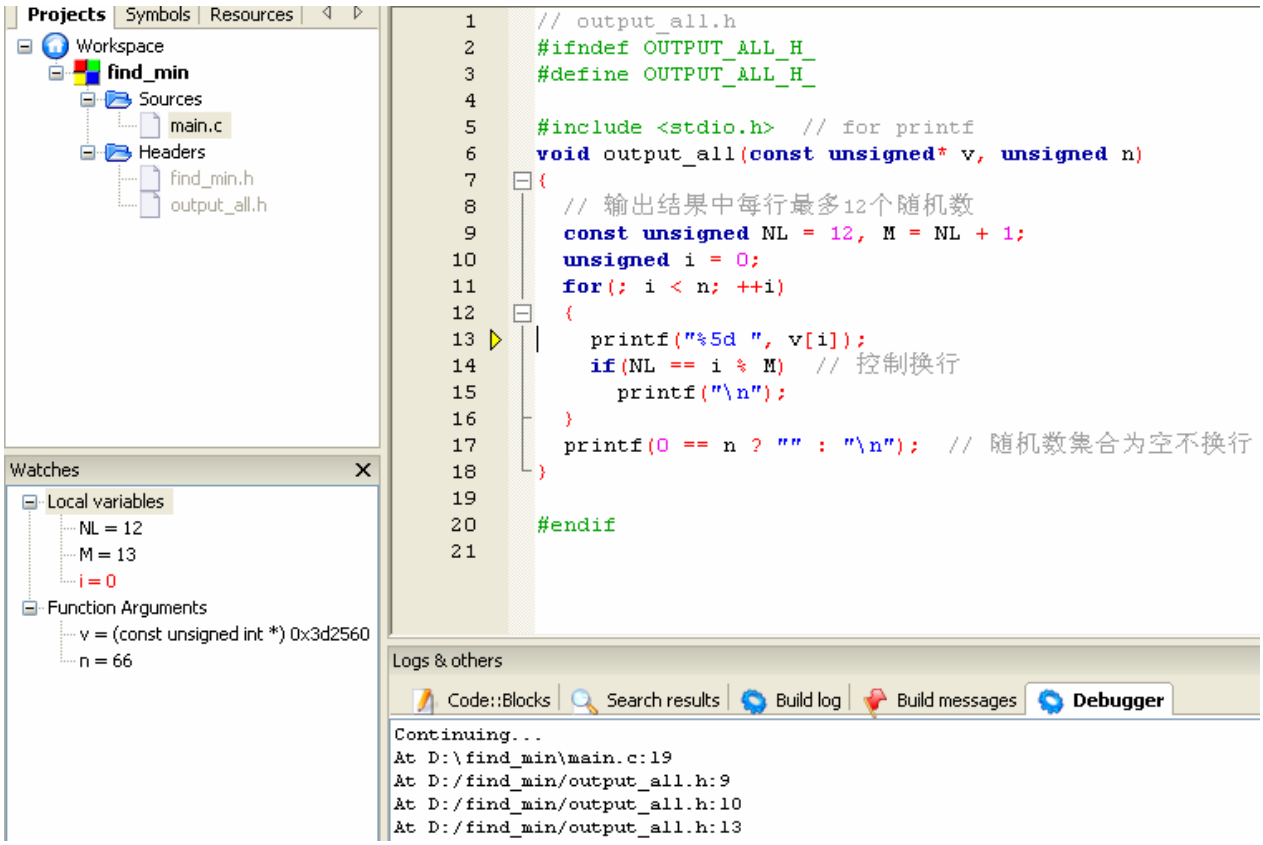


(6) 执行到output_all函数

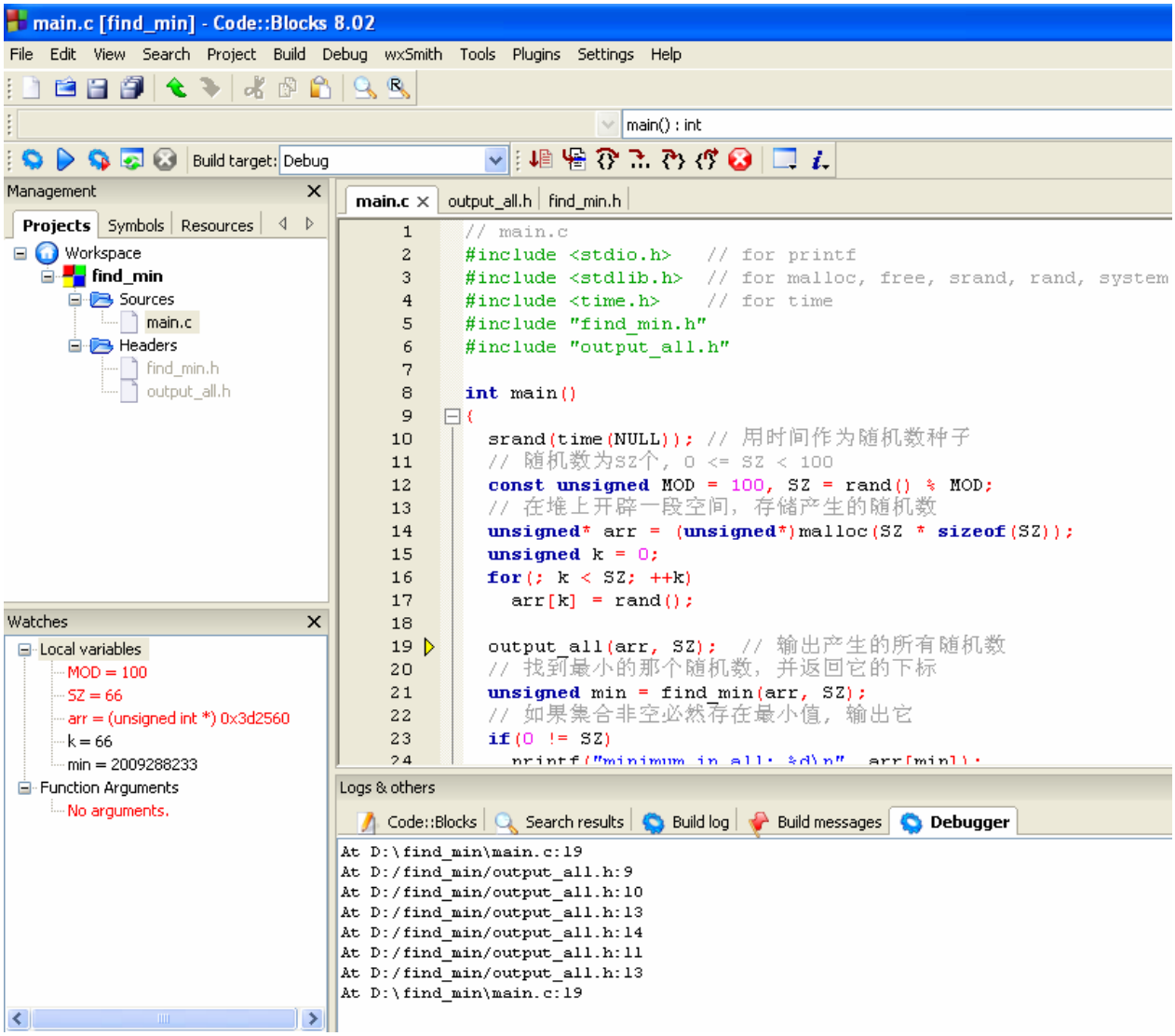
把光标置于第19行前，执行到此行，然后用鼠标点击Step into，则出现如下界面。



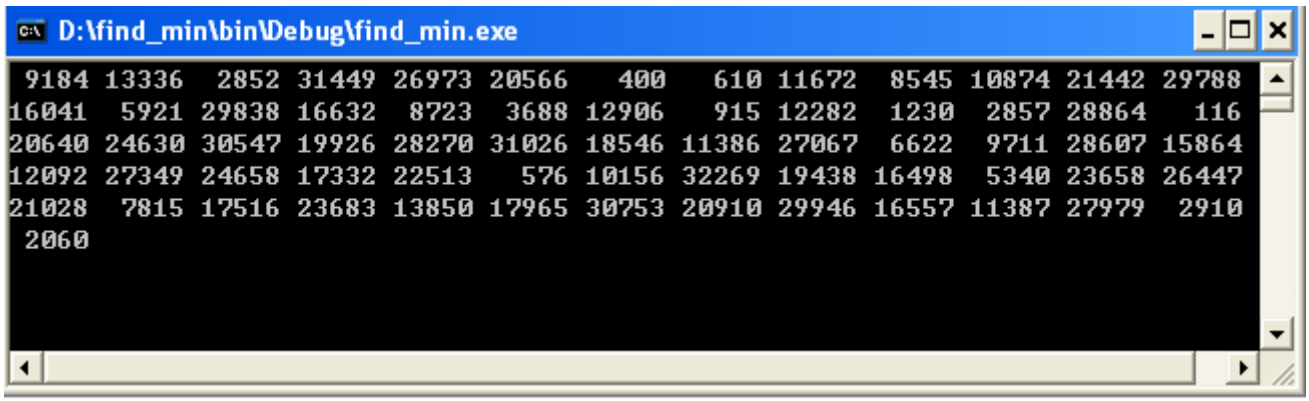
从上图可以看出，由于执行到函数output_all的第9行，函数参数v和n已经被赋值，但是NL, M, i三个局部变量都是系统赋予的随机值，但是执行到第13行前的时候，三个局部变量都已被正常赋值，见下图。



点击Step out按钮，则跳出output_all函数体，此时output_all函数已经执行完毕，见下图。



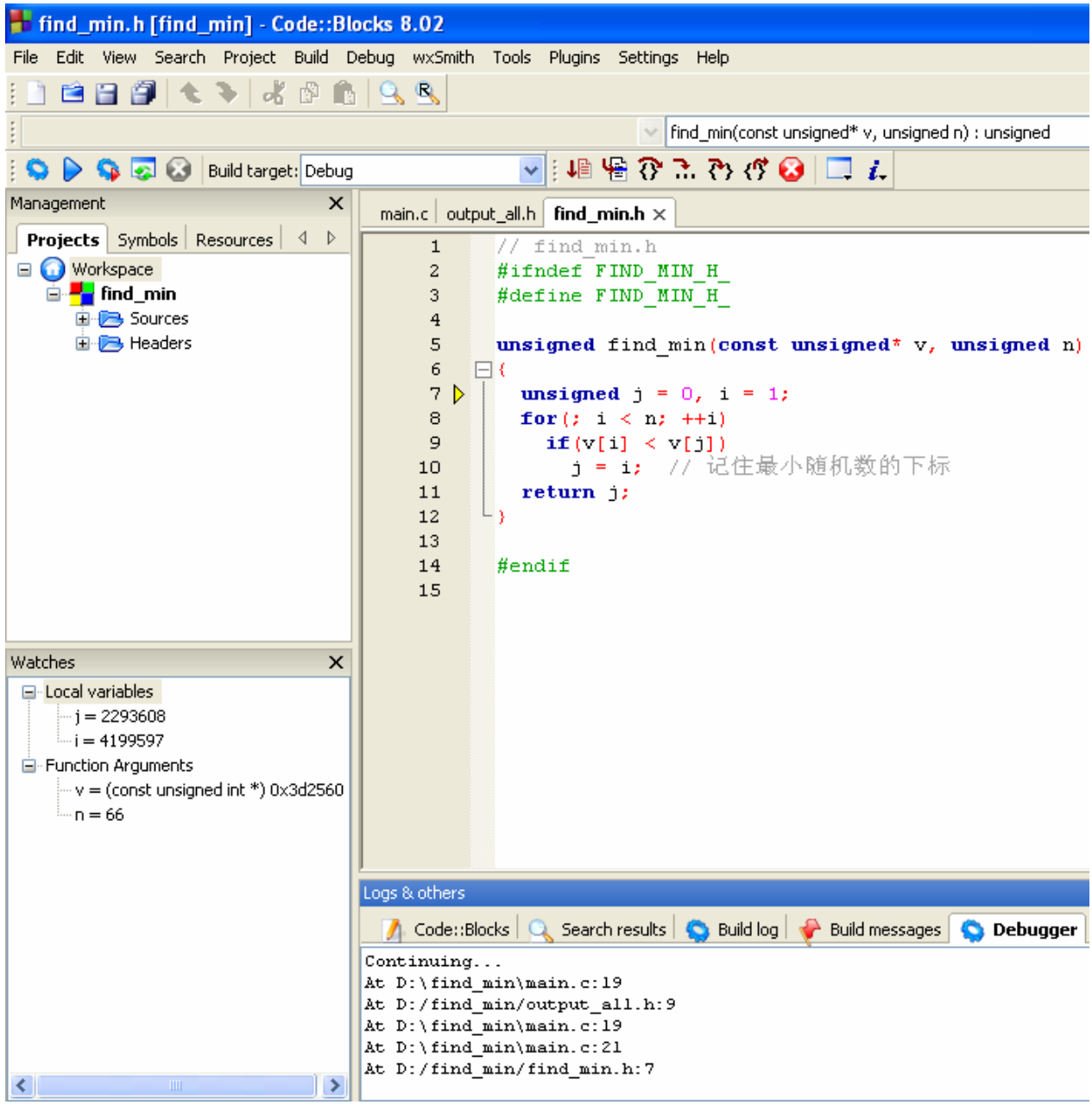
从上图日志Debugger栏目的信息可以看出，output_all函数已经执行完毕，重新进入main.c函数体，并已经执行完第19行代码，并且屏幕显示了一些信息(output_all函数刚执行完毕后的结果)，见下图。



(7) 执行到find_min函数

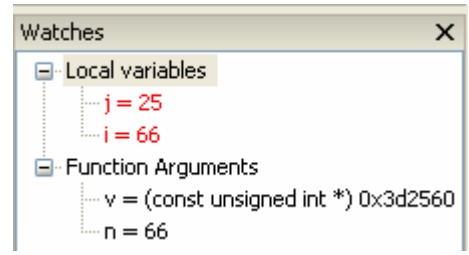
把光标置于main函数第21行前，然后点击Run to Cursor按钮，再点击Step into按钮，则进入

find_min函数体，见下图。

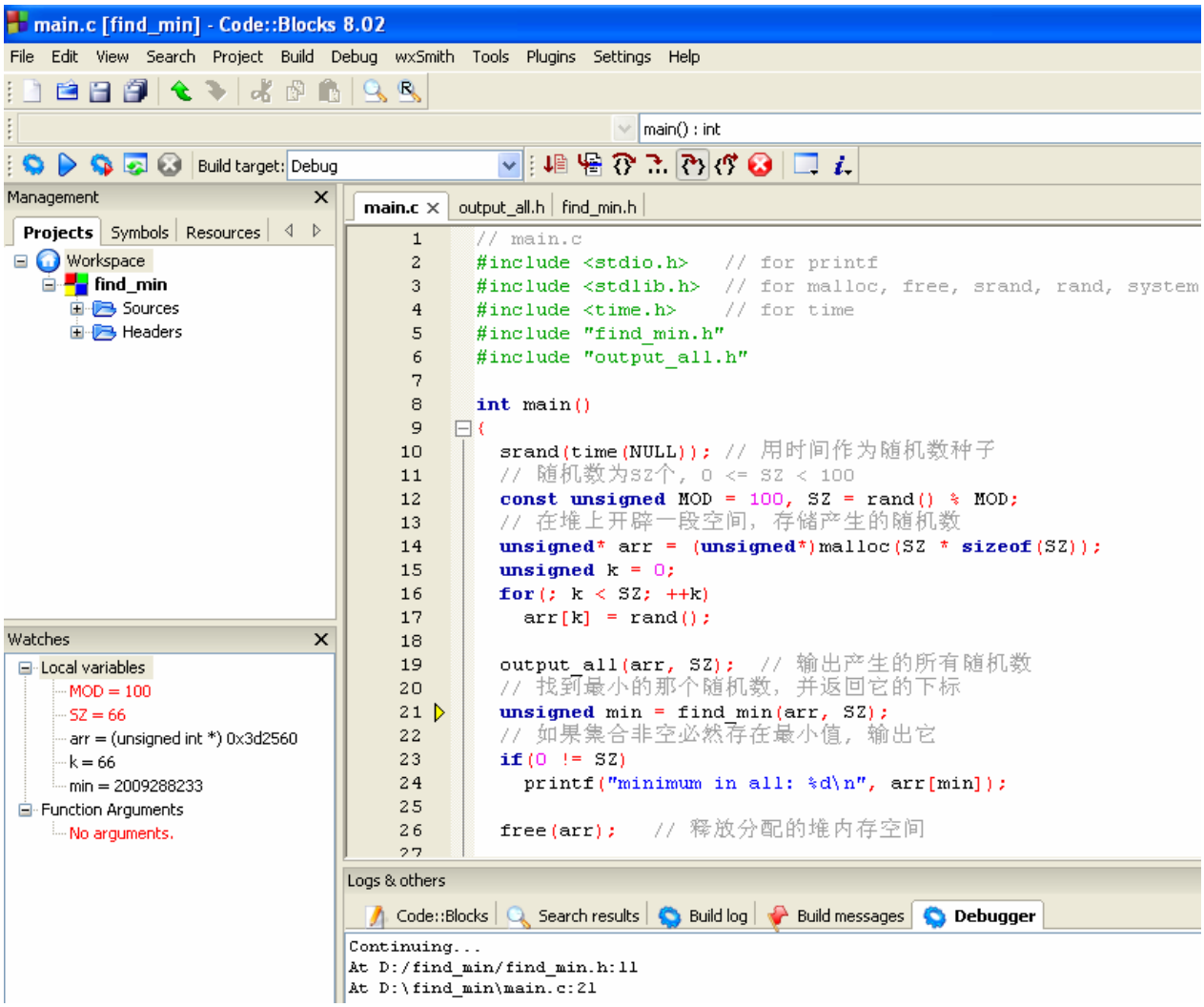


从上图可以看出，已经执行到find_min函数的第7行代码前，此时find_min函数的参数已经被赋值，但是局部变量j和i仍然是系统赋予的随机值(因为第7行代码尚未执行)。

把光标置于第11行代码前，点击Run to Cursor按钮，此时局部变量j已经得到确定值25(最小值下标)，由于循环体已经执行完毕，因此i得到66(跟传递进来的数组大小值相同，等于该次n的值)，见右边Watches窗口的截图。

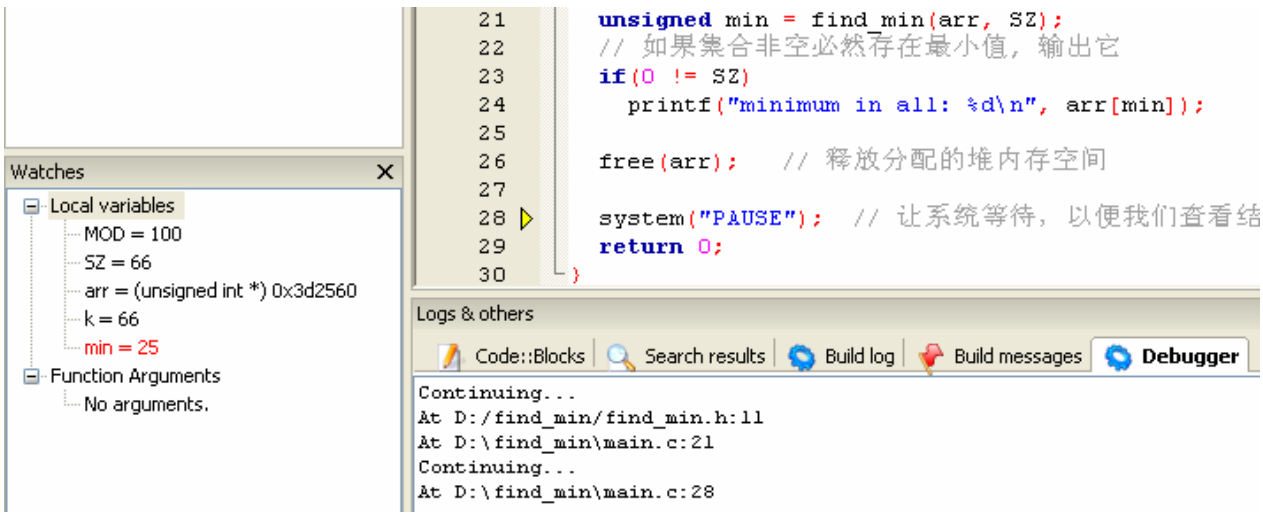


点击Step out按钮，则跳出find_min函数体，重新进入main函数，见下图。

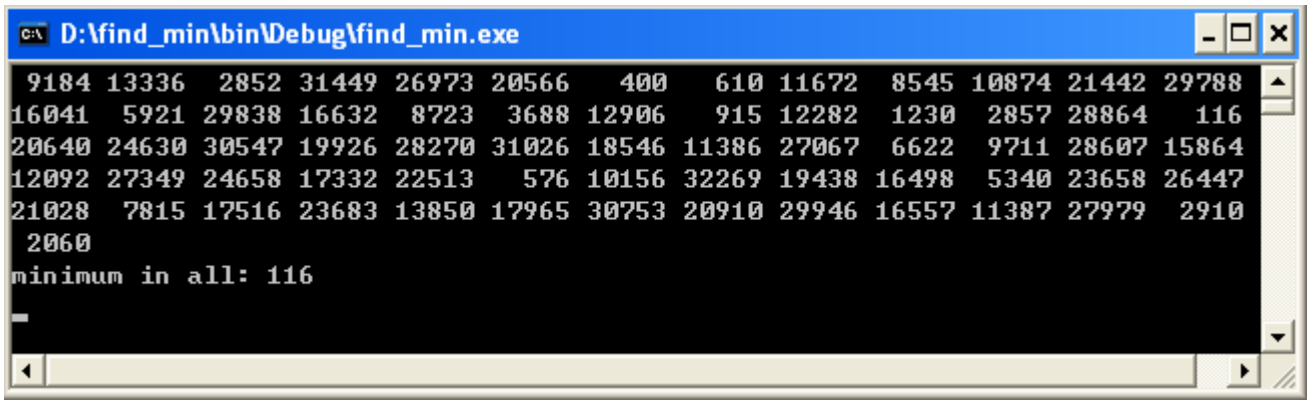


(8) 输出最小值

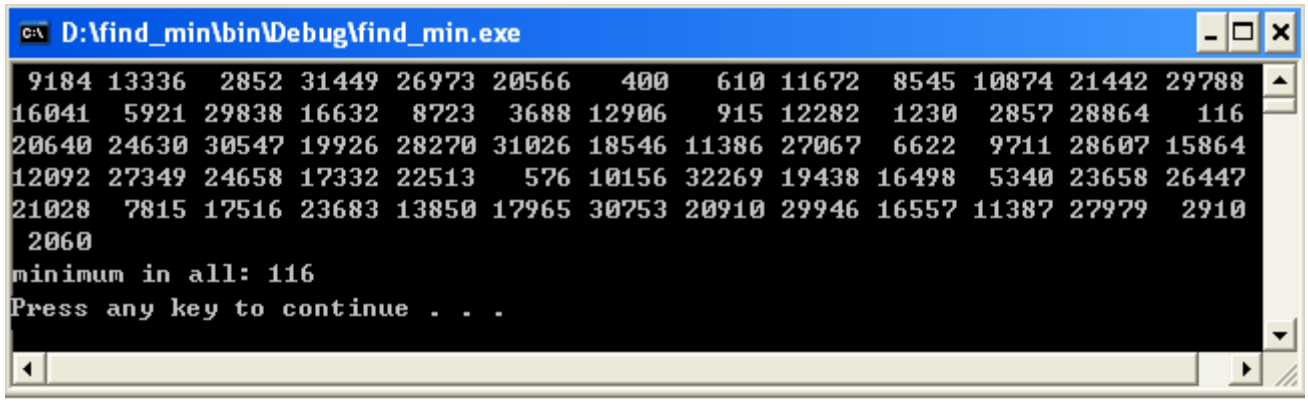
把光标置于第28行代码前面，点击Run to Cursor按钮，可以看到Watches窗口已经显示局部变量min的值为25，见下图。



由于main函数同时输出信息窗口也显示了最小值，见下图。



用鼠标点击Continue按钮继续运行，则输出如下信息，见下图(如果您使用的中文版操作系统，最下一行信息可能是：按任意键继续．．．)。



程序运行完毕，用鼠标点击Stop debugger按钮关闭调试器即可。以上程序中也可以不用在堆上动态分配空间(见main函数第14行代码)，由于需要的空间不太大，可以使用数组在栈上分配空间即可。

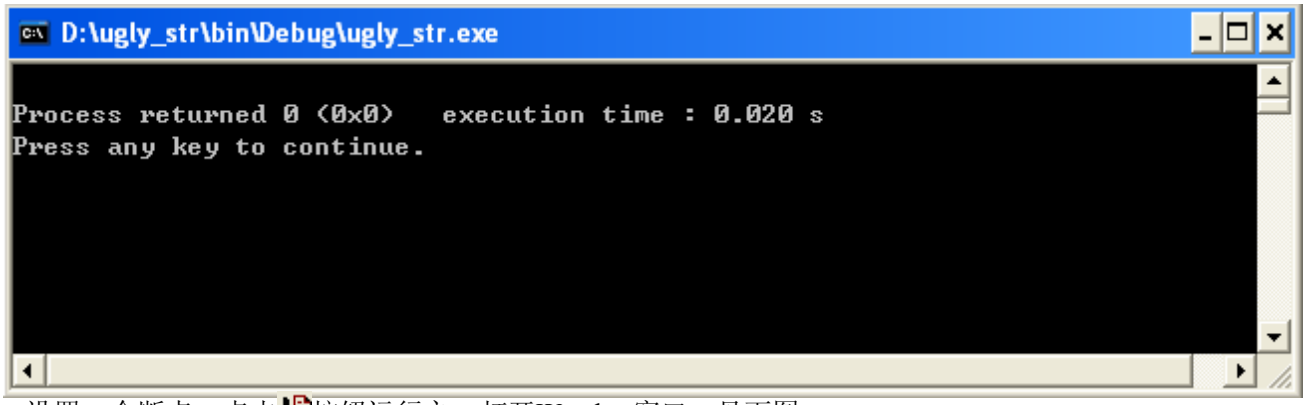
本小结例子中的程序编译运行结果正确，没有逻辑错误，这样的程序无需调试，只不过为了说明怎么使用gdb调试器而进行调试的。

3.5.2 面向对象风格的程序

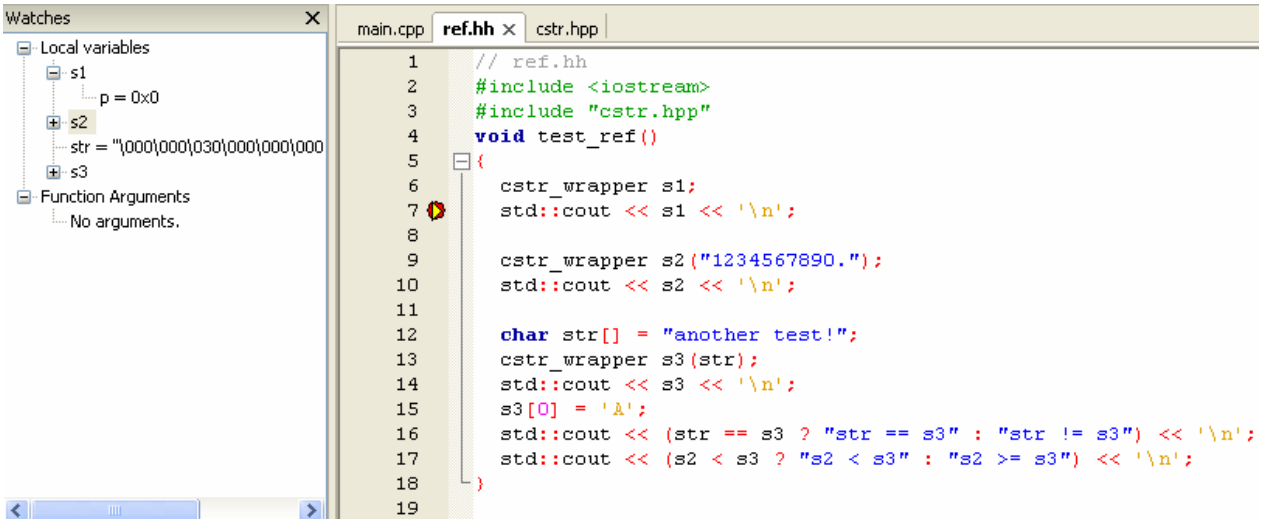
以下有一个cstr_wrapper的类，用来封装C风格的字符窜，仅仅对C风格的字符窜进行引用(不做深层复制)，提供了一些必要的成员函数。本程序中不止一处有逻辑错误，现在使用gdb调试器来找出错误的位置和原因。

建立一个工程，取名为ugly_str(取这个名字的原因很简单，因为这样设计封装C风格字符窜很愚蠢)，该工程中有四个文件，main.cpp, cstr.hpp, ref.hh, copy.hh。其中cstr.hpp定义了cstr_wrapper, 对C风格字符窜进行封装，ref.hh和copy.hh是两个测试函数，分别测试引用和复制字符窜时是否工作正常。建立的这个工程跟上小节3.5.1中共用一个Workspace(工作空间名字随便取)，编译或者调试这个工程前需要激活它(让系统知道你对它操作，而不是对原来的find_min工程操作)，激活的方法很简单，选中工程名ugly_str，按下鼠标右键弹出的菜单中选择Activate project，则该工程此时就处于激活状态，原来激活状态的find_min工程进入休眠。

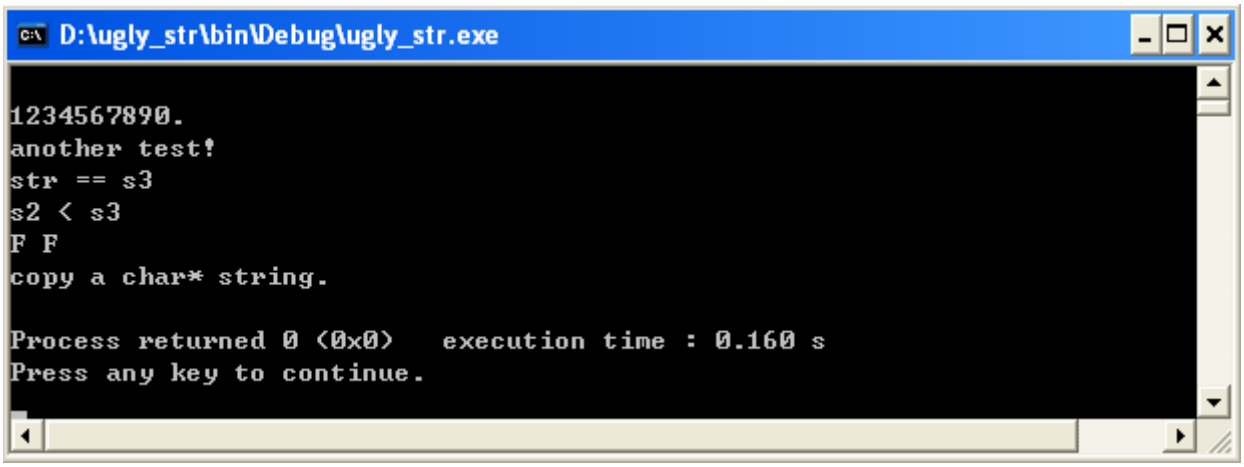
编译并运行之，见下面弹出的窗口，居然没有显示任何信息！



设置一个断点，点击按钮运行之，打开Watches窗口，见下图。



从上图中可以看出，已经建立对象s1，但是它的值(指针)为0，不指向字符窜。在32位Windows系统上，操作系统占用低地址，地址为0的内存禁止用户程序访问，访问系统占用内存是非法操作，如果您懂gcc汇编的话，打开汇编窗口，或者用鼠标点击Next instruction按钮就可以看到汇编代码，更容易看出问题。查看cstr_wrapper的构造函数发现，有这样一个函数头cstr_wrapper(const char* str = NULL)；显然这里有问题，空窜应该是""，而不是NULL，改正过来重新编译运行，结果如下。



从上图可以看出ref.hh函数测试已经通过，但是copy.hh测试输出的第一个字符窜就错了，结合源代

码，先注释掉test_ref();再对test_copy();进行跟踪。

程序代码见下面的贴图。

```cpp
// main.cpp
#include "copy.hh"
#include "ref.hh"

int main()
{
    test_ref();
    test_copy();
    return 0;
}
```

```cpp
// copy.hh
#include "cstr.hpp"
cstr_wrapper test_copy1()
{
    char str[] = "copy a char[] string.";
    return cstr_wrapper(str);
}

cstr_wrapper test_copy2()
{
    char* str = "copy a char* string.";
    return cstr_wrapper(str);
}

#include <iostream>
void test_copy()
{
    cstr_wrapper str1 = test_copy1();
    std::cout << str1 << ' ';
    str1[0] = 'C';
    std::cout << str1 << '\n';

    cstr_wrapper str2 = test_copy2();
    std::cout << str2 << '\n';
}
```

```cpp
// cstr.hpp
#ifndef CSTR_WRAPPER_HPP_
#define CSTR_WRAPPER_HPP_

#include <cstring>
#include <iostream>
#include <stdexcept>

class cstr_wrapper
{
public:
    cstr_wrapper(const char* str = NULL);
    cstr_wrapper(const cstr_wrapper&);

    unsigned size()const;              // 字符窜中的字符个数，不包括结束符
    bool empty()const;                 // 判断是否为空窜
    char& operator [] (unsigned);      // 引用某个字符，不检查下标
    const char& operator [] (unsigned) const;
    const char& at(unsigned) const;    // 引用某个字符，检查下标
    const char* c_str() const;         // 转换成c风格的字符窜
    // 其它成员函数，略...
    // 字符窜比较，重载<和==
    friend bool operator < (const cstr_wrapper&, const cstr_wrapper&);
    friend bool operator == (const cstr_wrapper&, const cstr_wrapper&);
    // 输出字符窜，重载<<
    friend std::ostream& operator << (std::ostream&, const cstr_wrapper&);

private:
    char* p;
};
```

```
31
32  // 构造函数
33  cstr_wrapper::cstr_wrapper(const char* str) : p(const_cast<char*>(str)) {}
34  cstr_wrapper::cstr_wrapper(const cstr_wrapper& s) : p(s.p) {}
35  //字符串大小，以及取某个字符(不检查下标)
36  unsigned cstr_wrapper::size() const { return std::strlen(p); }
37  bool cstr_wrapper::empty() const { return 0 == std::strcmp(p, ""); }
38  const char& cstr_wrapper::operator [] (unsigned i) const { return p[i]; }
39  char& cstr_wrapper::operator [] (unsigned i) { return p[i]; }
40  // 取某个字符时，对下标进行检查
41  const char& cstr_wrapper::at(unsigned i) const
42  {
43    if(i < std::strlen(p))
44      return p[i];
45    else
46      throw std::out_of_range("subscript error");
47  }
48  // 转换成c风格的字符串
49  const char* cstr_wrapper::c_str() const { return p; }
50  // 判断字符串大小
51  bool operator < (const cstr_wrapper& s1, const cstr_wrapper& s2)
52  {
53    return std::strcmp(s1.p, s2.p) < 0;
54  }
55  // 判断字符串是否相等
56  bool operator == (const cstr_wrapper& s1, const cstr_wrapper& s2)
57  {
58    return 0 == std::strcmp(s1.p, s2.p);
59  }
60  // 输出一个字符串
61  std::ostream& operator << (std::ostream& os, const cstr_wrapper& s)
62  {
63    return os << s.p;
64  }
65
```

```
1   // ref.hh
2   #include <iostream>
3   #include "cstr.hpp"
4   void test_ref()
5   {
6     cstr_wrapper s1;
7     std::cout << s1 << '\n';
8
9     cstr_wrapper s2("1234567890.");
10      std::cout << s2 << '\n';
11
12    char str[] = "another test!";
13    cstr_wrapper s3(str);
14    std::cout << s3 << '\n';
15    s3[0] = 'A';
16    std::cout << (str == s3 ? "str == s3" : "str != s3") << '\n';
17    std::cout << (s2 < s3 ? "s2 < s3" : "s2 >= s3") << '\n';
18  }
19
```

```cpp
// main.cpp
#include "copy.hh"
#include "ref.hh"

int main()
{
  //test_ref();
  test_copy();
  return 0;
}
```

```cpp
#include <iostream>
void test_copy()
{
  cstr_wrapper str1 = test_copy1();
  std::cout << str1 << ' ';
  str1[0] = 'C';
  std::cout << str1 << '\n';

  cstr_wrapper str2 = test_copy2();
  std::cout << str2 << '\n';
}
```

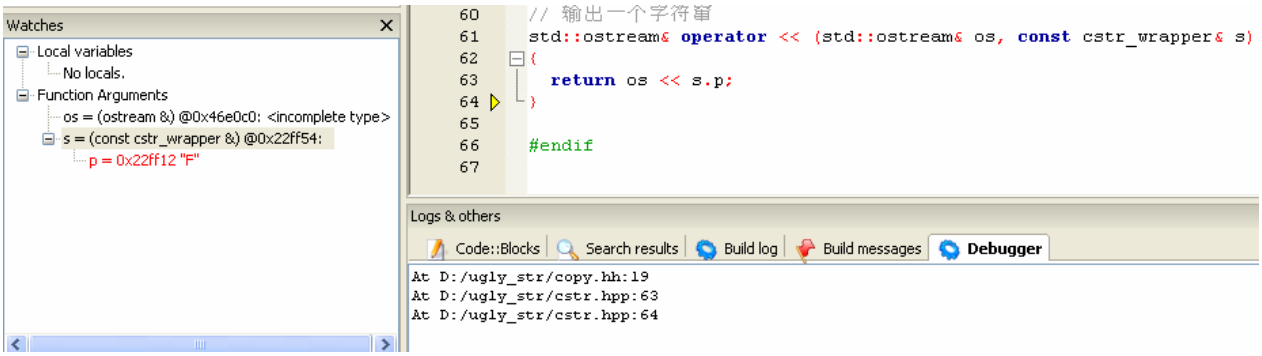
设置好断点，点击按钮启动调试器，见上图，打开Watches窗口，进行跟踪。点击Step into按钮，进入test_copy1的函数体，见下图。

```cpp
// copy.hh
#include "cstr.hpp"
cstr_wrapper test_copy1()
{
  char str[] = "copy a char[] string.";
  return cstr_wrapper(str);
}
```

继续Step into跟踪对象的创建过程，见下图。

```cpp
// 构造函数
cstr_wrapper::cstr_wrapper(const char* str) : p(const_cast<char*>(str)) {}
cstr_wrapper::cstr_wrapper(const cstr_wrapper& s) : p(s.p) {}
//字符窜大小，以及取某个字符(不检查下标)
unsigned cstr_wrapper::size() const { return std::strlen(p); }
bool cstr_wrapper::empty() const { return 0 == std::strcmp(p, ""); }
const char& cstr_wrapper::operator [] (unsigned i) const { return p[i]; }
char& cstr_wrapper::operator [] (unsigned i) { return p[i]; }
```

从上面的截图可以看出，调用cstr_wrapper(const char*);创建对象，再结合test_copy1函数的代码可以发现虽然返回一个临时对象的拷贝没有任何问题，但cstr_wrapper的定义中构造函数仅仅拷贝了指针，待test_copy1函数体内的str过期被销毁的时候，新创建的对象(仅仅一个指针)将指向何处？继续跟踪到调用重载operator<<的友元函数输出时，从Watches可以发现此时创建的对象值(指针)，已经指向一个莫名其妙的字符窜"F"(在您的电脑上可能是其他字符窜内容)。



由于cstr_wrapper本身定义就是复制一个指针，没有办法对它进行大的改动(除非重新设计和编码)，

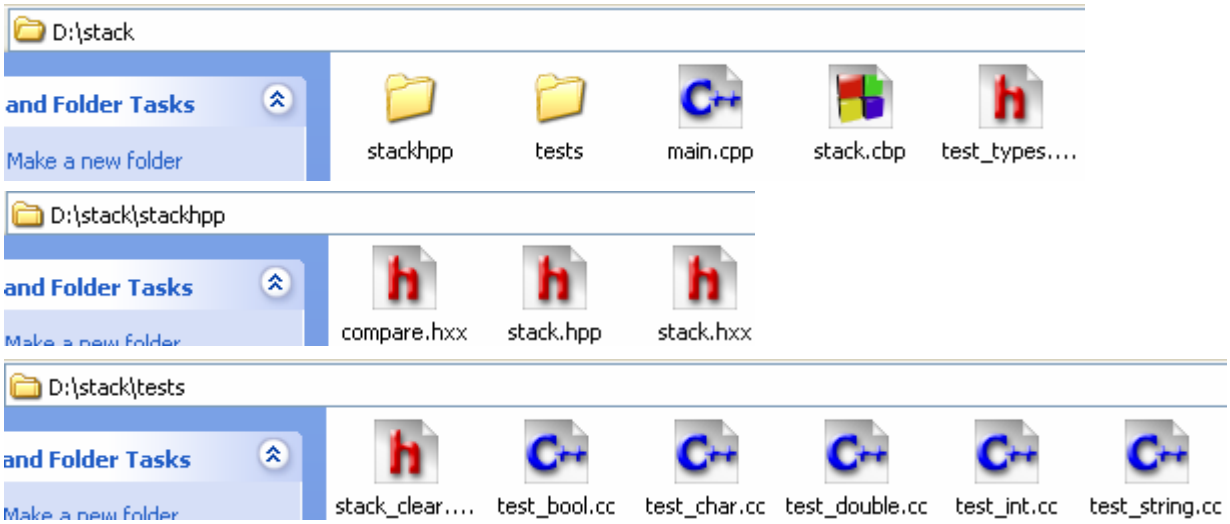
只能遗弃不用cstr_wrapper。

也许有的读者还想不明白为什么test_copy2函数能正常工作。虽然这个函数也是返回一个拷贝的对象(指针)，但是它的函数体内有一条语句char* str = “copy a char* string.”;，由于C++本身设计为了兼容C的语法，而C语言使用这种格式定义字符窜，在程序停止运行前不会被销毁，程序运行结束后由操作系统回收占用的资源，因此拷贝指向这个字符窜的指针在程序运行的生命周期内始终不会成为野指针，自然就能正常输出了。

3.5.3 泛型程序

泛型程序调试更复杂一点，因为一个函数或者类往往由于初始化类型的不同，生成不同种类的对象，而且函数和运算符重载过程也比一般的函数和运算符重载复杂的多。

以下是一个简易栈的实现以及几个运算符重载，为了确保该栈实现的健壮性，也对它进行一些必要的测试。下面是各文件目录层次关系。



下面是代码贴图。

```
1   // compare.hxx
2   #ifndef COMPARE_HXX_
3   #define COMPARE_HXX_
4
5   //比较任何可比的简单类型或者复杂类型(包括容器类型)大小
6   template <typename T>
7   bool operator != (const T& x, const T& y) { return !(x == y); }
8
9   template <typename T>
10  bool operator <= (const T& x, const T& y) { return !(y < x); }
11
12  template <typename T>
13  bool operator > (const T& x, const T& y) { return y < x; }
14
15  template <typename T>
16  bool operator >= (const T& x, const T& y) { return !(x < y); }
17
18  #endif
19
```

```cpp
// main.cpp
#include "test_types.hpp"
int main()
{
  test_types();
  return 0;
}

```

```cpp
// stack.hpp
#ifndef STACK_HPP_
#define STACK_HPP_

#include "compare.hxx"
#include "stack.hxx"

#endif

```

```cpp
// stack.hxx
#ifndef STACK_HXX_
#define STACK_HXX_

#include <iostream>
#include <stdexcept>
#include <cstdlib>
#include <cstring>

template <class T>                              // T必须是POD类型
class Stack
{
public:
  typedef unsigned long size_type;
  typedef T& reference;
  typedef const T& const_reference;
public:
  Stack() : cap(SZ), sz(0) { data = get(cap); }
  Stack(unsigned n, const T& d = T()) : cap(SZ), sz(n)
  {
    while(cap < sz) cap <<= 1;
    data = get(cap);
    std::memset(data, d, tsize() * sz);
  }
  Stack(const Stack& st) : cap(st.cap), sz(st.sz), data(get(cap))
  {
    std::memcpy(data, st.data, sz * tsize());
  }
  ~Stack()
  {
    delete [] data;
    sz = cap = 0;
  }

  Stack& operator = (const Stack& s)        // 可以赋值
  {
    if(this != &s)
    {
      delete [] data;
      cap = s.cap;
      data = get(cap);
      sz = s.sz;
      std::memcpy(data, s.data, sz * tsize());
    }
    return *this;
  }
```

```
47      unsigned size() const  { return sz; }    // 栈中元素个数
48      bool empty() const  { return 0 == sz; } // 栈是否空
49      const T& top()const                      // 栈顶元素
50      {
51        if (sz < 1) Underflow_Error();
52        return data[sz - 1];
53      }
54      T& top()
55      {
56        if(sz < 1) Underflow_Error();
57        return data[sz - 1];
58      }
59      void push(const T& d)                    // 入栈
60      {
61        if(sz + 1 > cap)
62        rebuild();
63        data[sz++] = d;
64      }
65      void pop()                               // 删除栈顶元素
66      {
67        if(sz < 1) Underflow_Error();
68        --sz;
69      }
70
71      bool operator == (const Stack& st) const
72      {
73        return 0 == std::memcmp(data, st.data, sz * tsize());
74      }
75
76      bool operator < (const Stack& st) const
77      {
78        return std::memcmp(data, st.data, sz * tsize()) < 0;
79      }
80      void swap(Stack& s)                      // 交换两个栈
81      {
82        unsigned t;
83        t = cap, cap = s.cap, s.cap = t;
84        t = sz, sz = s.sz, s.sz = t;
85        T* d;
86        d = data, data = s.data, s.data = d;
87      }
88    protected:
89      T* get(unsigned n)                       // 分配内存
90      {
91        T* p;
92        try { p = new T[n]; }
93        catch(const std::bad_alloc& ba)
94        {
95          std::cerr << ba.what();
96          throw;
97        }
98        return p;
99      }
```

```
100
101     void rebuild()                          // 栈扩大为原来的2倍
102     {
103       cap <<= 1;
104       T* p = get(cap);
105       std::memcpy(p, data, sz * tsize());
106       delete [] data;
107       data = p;
108     }
109
110   private:
111     void Underflow_Error() { throw std::underflow_error("Stackunderflowed"); }
112     unsigned tsize() const { return sizeof(T); }
113     static const unsigned SZ = 1;
114     unsigned cap, sz;
115     T* data;
116   };
117
118   #endif
119
```

```
 1   // test_types.hpp
 2   #include <cstdlib>
 3   #include <ctime>
 4
 5   #include "tests/test_int.cc"
 6   #include "tests/test_double.cc"
 7   #include "tests/test_char.cc"
 8   #include "tests/test_bool.cc"
 9   #include "tests/test_string.cc"
10
11   static void init_rand() { std::srand(std::time(0)); }
12   void test_types()
13   {
14     init_rand();
15
16     test_int();
17     test_double();
18     test_char();
19     test_bool();
20     test_string();
21   }
```

```
 1   // stack_clear.hpp
 2   #ifndef STACK_CLEAR_HPP_
 3   #define STACK_CLEAR_HPP_
 4
 5   #include "../stackhpp/stack.hpp"
 6
 7   // 清空一个栈
 8   template <typename T>
 9   void stack_clear(Stack<T>& s)
10   {
11     if(!s.empty())
12     {
13     Stack<T> st; // 建立一个临时空栈
14     st.swap(s);  // s存储st的元素
15     } // st已经被销毁
16   }
17
18   #endif
19
```

```cpp
// test_string.cc
#include <cstdlib>
#include <string>
#include <iostream>
#include "stack_clear.hpp"

void rand_str(std::string& str) // 产生随机字符串，仅仅是字母和数字的组合
{
  const std::string
    carr("0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz");
  for(std::string::size_type i = std::rand() % 75; i > 0; --i)
    str.push_back(carr[std::rand() % carr.size()]);
}
void test_string() // 测试stack<std::string>各个成员函数以及比较运算符
{
  typedef Stack<std::string>::size_type size_type;

  std::cout << "test strings:\n";
  //std::srand(std::time(0));
  const size_type M = 11, SZ1 = rand() % M, SZ2 = std::rand() % M;
  Stack<std::string> sts1, sts2;
  std::string str;
  // 用随机字符串填充stack<std::string> sts1
  for(unsigned i = 0; i < SZ1; ++i)
  {
    rand_str(str);
    sts1.push(str);
    str.clear();
  }

  for(unsigned i = 0; i < SZ2; ++i)
  {
    rand_str(str);
    sts2.push(str);
    str.clear();
  }

  std::cout << "size of Stack: " << sts1.size() << ' ' << sts2.size() << '\n';
  std::cout << "top element: " << (sts1.empty() ? "" : sts1.top()) << '\n';
  std::cout << "top element: " << (sts2.empty() ? "" : sts2.top()) << '\n';
  std::cout << "comparing Stack sts1 and sts2:\n";
  std::cout << (sts1 != sts2 ? "sts1 != sts2" : "sts1 == sts2") << '\n';
  std::cout << (sts1 > sts2 ? "sts1 > sts2" : "sts1 <= sts2") << '\n';
  std::cout << (sts1 >= sts2 ? "sts1 >= sts2" : "sts1 < sts2") << '\n';
  sts2.swap(sts1);
  std::cout << (sts1 <= sts2 ? "sts1 <= sts2" : "sts1 > sts2") << '\n';

  stack_clear(sts1);
  stack_clear(sts2);
  std::cout << "strings test ok!\n\n";
}

```

```cpp
// test_int.cc
#include <cstdlib>
#include <iostream>
#include "stack_clear.hpp"

void test_int() // 测试stack<int>各个成员函数以及比较运算符
{
  typedef Stack<int>::size_type size_type;

  std::cout << "test integers:\n";
  //std::srand(std::time(0));
  const size_type M = 11, SZ1 = std::rand() % M, SZ2 = std::rand() % M;
  Stack<int> sti1, sti2;
  // 用随机数填充Stack<int> sti1
  for(size_type i = 0; i < SZ1; ++i)
    sti1.push(std::rand());
  for(size_type i = 0; i < SZ2; ++i)
    sti2.push(std::rand());

  std::cout << "size of Stack: " << sti1.size() << ' ' << sti2.size() << '\n';
  std::cout << "top element: ";
  if(!sti1.empty())
    std::cout << sti1.top() << ' ';
  if(!sti2.empty())
    std::cout << sti2.top() << '\n';
  std::cout << "comparing Stack sti1 and sti2:\n";
  std::cout << (sti1 != sti2 ? "sti1 != sti2" : "sti1 == sti2") << '\n';
  std::cout << (sti1 > sti2 ? "sti1 > sti2" : "sti1 <= sti2") << '\n';
  std::cout << (sti1 >= sti2 ? "sti1 >= sti2" : "sti1 < sti2") << '\n';
  sti2.swap(sti1);
  std::cout << (sti1 <= sti2 ? "sti1 <= sti2" : "sti1 > sti2") << '\n';

  stack_clear(sti1);
  stack_clear(sti2);
  std::cout << "integers test ok!\n\n";
}
```

```cpp
// test_bool.cc
#include <cstdlib>
#include <iostream>
#include "stack_clear.hpp"

void test_bool() // 测试stack<bool>成员函数以及比较运算符
{
  typedef Stack<bool>::size_type size_type;

  std::cout << "test booleans:\n";
  const size_type M = 11, SZ1 = std::rand() % M, SZ2 = std::rand() % M;
  Stack<bool> stb1, stb2;
  // 用bool值true或者false填充stb1
  for(size_type i = 0; i < SZ1; ++i)
    stb1.push(std::rand() & 1);
  for(size_type i = 0; i < SZ2; ++i)
    stb2.push(std::rand() & 1);
```

```
18
19      std::cout << "size of Stack: " << stb1.size() << ' ' << stb2.size() << '\n';
20      std::cout << "top element: ";
21      std::cout << (stb1.empty() ? "" : (stb1.top() == true ? "true" : "false")) << ' ';
22      std::cout << (stb2.empty() ? "" : (stb2.top() == true ? "true" : "false")) << '\n';
23      std::cout << "comparing Stack stb1 and stb2:\n";
24      std::cout << (stb1 != stb2 ? "stb1 != stb2" : "stb1 == stb2") << '\n';
25      std::cout << (stb1 > stb2 ? "stb1 > stb2" : "stb1 <= stb2") << '\n';
26      std::cout << (stb1 >= stb2 ? "stb1 >= stb2" : "stb1 < stb2") << '\n';
27      stb2.swap(stb1);
28      std::cout << (stb1 <= stb2 ? "stb1 <= stb2" : "stb1 > stb2") << '\n';
29
30      stack_clear(stb1);
31      stack_clear(stb2);
32      std::cout << "booleans test ok!\n\n\n";
33  }
34
```

```
1   // test_double.cc
2   #include <cstdlib>
3   #include <iostream>
4   #include "stack_clear.hpp"
5
6   void test_double() // 测试stack<double>各个成员函数以及比较运算符
7   {
8     typedef Stack<double>::size_type size_type;
9
10    std::cout << "test real numbers:\n";
11    //std::srand(std::time(0));
12    const size_type M = 11, SZ1 = std::rand() % M, SZ2 = std::rand() % M;
13    Stack<double> std1, std2;
14    // 用随机浮点数填充Stack<double> std1
15    for(size_type i = 0; i < SZ1; ++i)
16      std1.push(static_cast<double>(std::rand()) / RAND_MAX);
17    for(size_type i = 0; i < SZ2; ++i)
18      std2.push(static_cast<double>(std::rand()) / RAND_MAX);
19
20    std::cout << "size of Stack: " << std1.size() << ' ' << std2.size() << '\n';
21    std::cout << "top element: ";
22    if(!std1.empty())
23      std::cout << std1.top()<< ' ';
24    if(!std2.empty())
25      std::cout << std2.top() << '\n';
26    std::cout << "comparing Stack std1 and std2:\n";
27    std::cout << (std1 != std2 ? "std1 != std2" : "std1 == std2") << '\n';
28    std::cout << (std1 > std2 ? "std1 > std2" : "std1 <= std2") << '\n';
29    std::cout << (std1 >= std2 ? "std1 >= std2" : "std1 < std2") << '\n';
30    std2.swap(std1);
31    std::cout << (std1 <= std2 ? "std1 <= std2" : "std1 > std2") << '\n';
32
33    stack_clear(std1);
34    stack_clear(std2);
35    std::cout << "floating points test ok!\n\n";
36  }
37
```

```cpp
// test_char.cc
#include <cstdlib>
#include <iostream>
#include "stack_clear.hpp"

void test_char()
{
  typedef Stack<char>::size_type size_type;

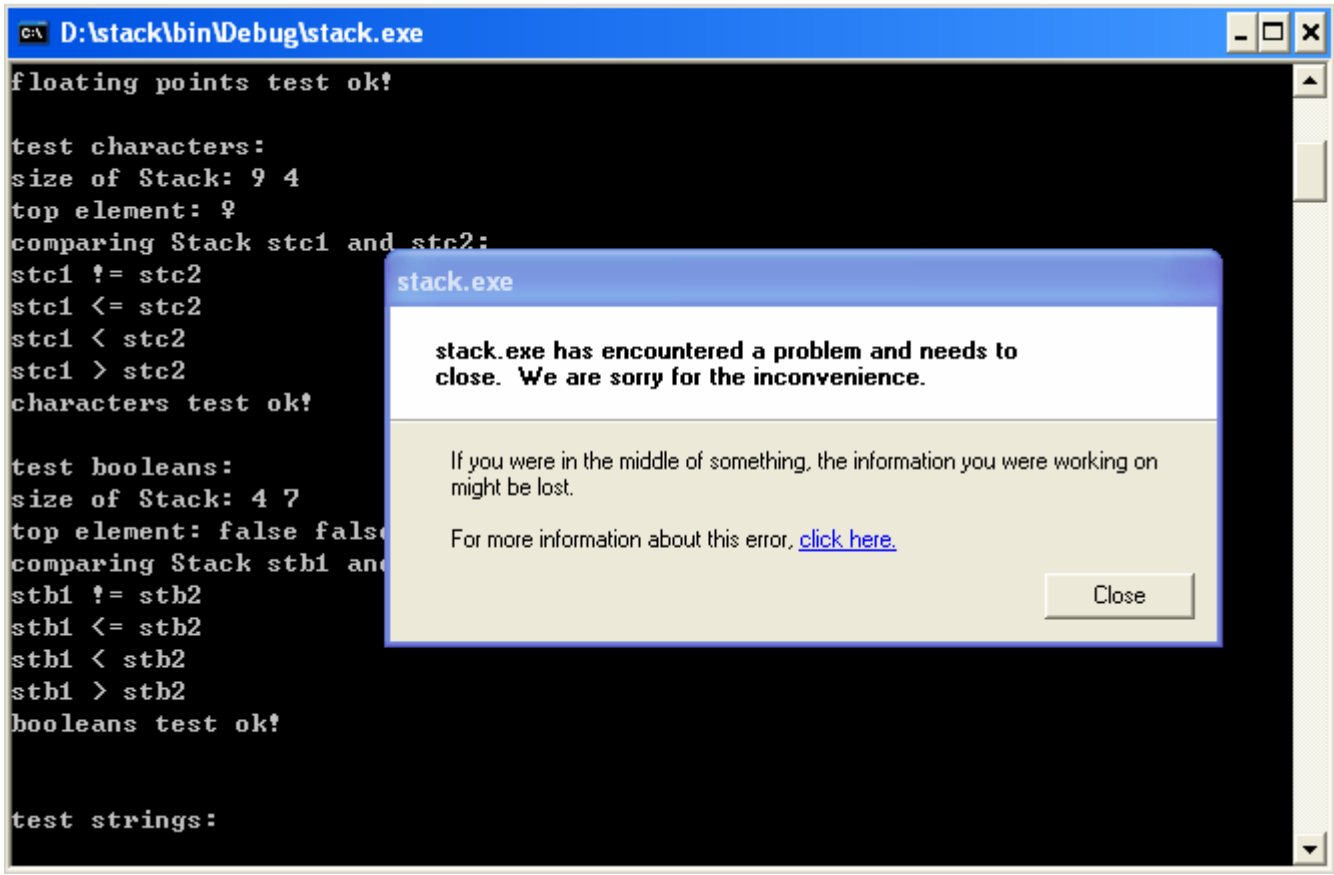
  std::cout << "test characters:\n";
  //std::srand(std::time(0));
  const size_type M = 11, SZ1 = std::rand() % M, SZ2 = std::rand() % M;
  Stack<char> stc1, stc2;
  // 用随机字符填充Stack<char> stc1
  for(size_type i = 0; i < SZ1; ++i)
    stc1.push(static_cast<char>(std::rand()));
  for(size_type i = 0; i < SZ2; ++i)
    stc2.push(static_cast<char>(std::rand()));

  std::cout << "size of Stack: " << stc1.size() << ' ' << stc2.size() << '\n';
  std::cout << "top element: ";
  std::cout << (stc1.empty() ? '\0' : stc1.top()) << ' ';
  std::cout << (stc2.empty() ? '\0' : stc2.top()) << '\n';
  std::cout << "comparing Stack stc1 and stc2:\n";
  std::cout << (stc1 != stc2 ? "stc1 != stc2" : "stc1 == stc2") << '\n';
  std::cout << (stc1 > stc2 ? "stc1 > stc2" : "stc1 <= stc2") << '\n';
  std::cout << (stc1 >= stc2 ? "stc1 >= stc2" : "stc1 < stc2") << '\n';
  stc2.swap(stc1);
  std::cout << (stc1 <= stc2 ? "stc1 <= stc2" : "stc1 > stc2") << '\n';

  stack_clear(stc1);
  stack_clear(stc2);
  std::cout << "characters test ok!\n\n";
}

```

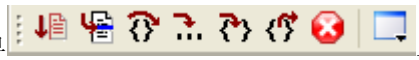
编译并运行该程序，运行时失败，弹出错误信息见下图。从显示结果看，是string测试出了问题。当然，很多人可能早就知道很多C的库函数(例如memcpy, memset等)，不能对对象操作，否则程序很容易崩溃，但是知道为什么吗？如果您并不清楚为什么，那就通过一个简单的例子，来看一看到底究竟怎么回事。

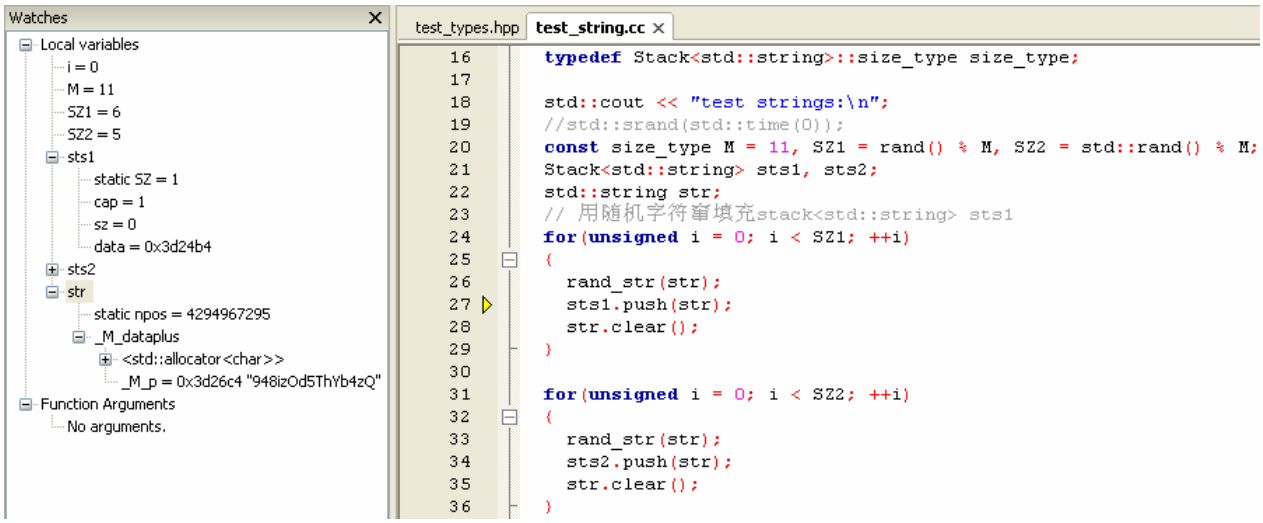


首先，注释掉test_types.hpp中跟test_string();无关的代码，因为其它几个数据类型测试已经通过，不必再跟踪。设置断点，然后启动调试器，见下面贴图。

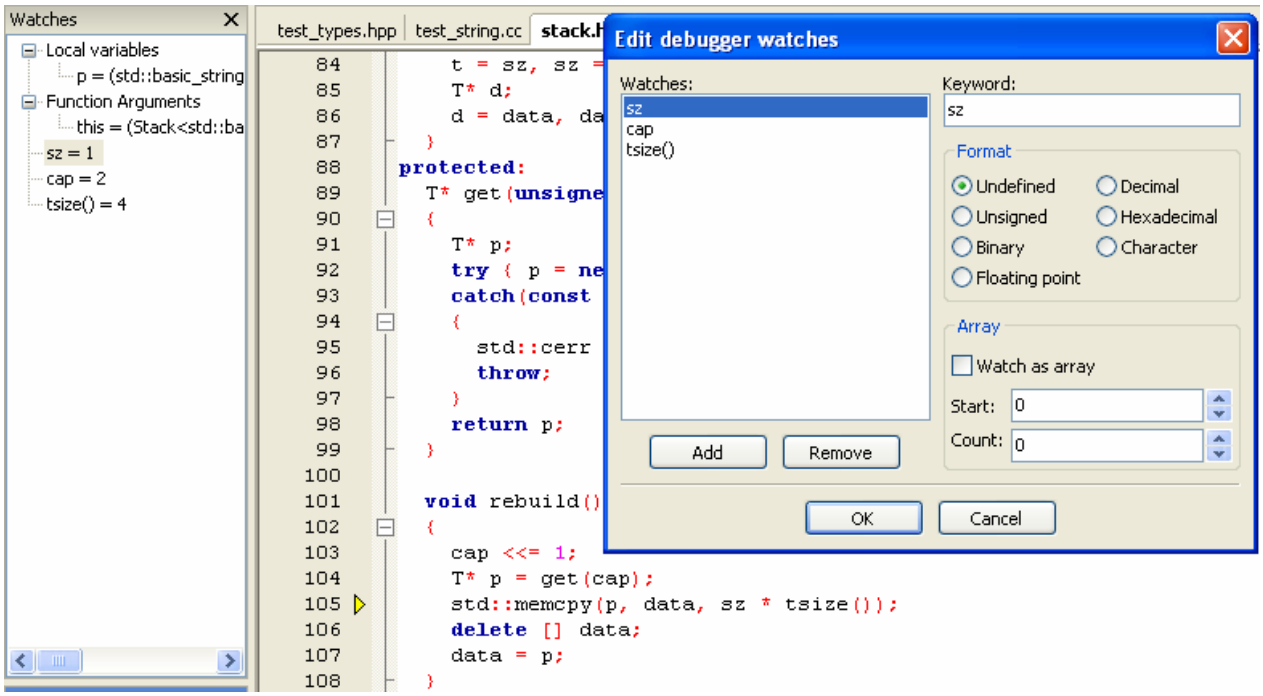
```
1   // test_types.hpp
2   #include <cstdlib>
3   #include <ctime>
4
5   //#include "tests/test_int.cc"
6   //#include "tests/test_double.cc"
7   //#include "tests/test_char.cc"
8   //#include "tests/test_bool.cc"
9   #include "tests/test_string.cc"
10
11  static void init_rand() { std::srand(std::time(0)); }
12  void test_types()
13  {
14    init_rand();
15
16  //  test_int();
17  //  test_double();
18  //  test_char();
19  //  test_bool();
20    test_string();
21  }
22
```

打开watches窗口，接下来专门针对string类型的Stack建立过程一路跟踪下去(注意巧妙的用好调试程

序的快捷菜单上的几个按钮)，已经跟踪到了test_string.cc第27行代码前，仍然没有问题，见下图。

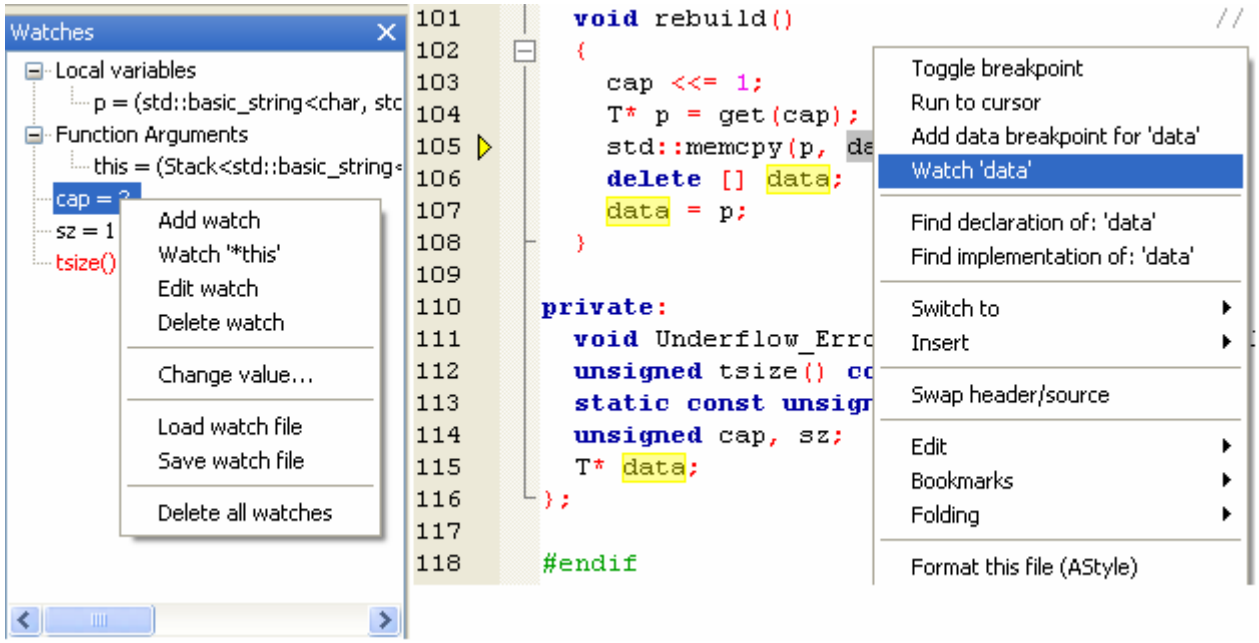


继续Step into，进入栈的push成员函数体内，压入栈第一个string对象没有发生任何问题，继续跟踪把第二个string对象压入栈的过程，此时栈空间不足，需要重新分配空间，然后把原来栈中的string对象复制到新开辟的空间再销毁原来的空间。这个过层中开辟空间没有问题，因为new足够智能，它知道该开辟多大的空间(因为new能调用构造函数创建一个string数组，T被初始化为new以后，再调用就成了p = new std::string[n])，push对象到栈中，这个复制过程也不会产生问题(因为std::string已经重载了=，可以正常复制std::string对象)，但是由于栈空间不足，调用rebuild函数时，我们从Debug下拉菜单选择Edit watches…增加几个变量，监测它们的变化，见下面的贴图。



编辑Waches窗口变量的方法很多，也可以把鼠标指示移动到Watches窗口内，按下鼠标右键会弹出

一个快捷菜单，有几个功能供我们选择。Add watch用于手工添加变量到Watches窗口内，Edit watch用来编辑Watches窗口内当前选中的变量，Delete watch则用来把选中的变量从Watches窗口内删除，见左下图。



如果想添加一个变量到Watches窗口，还有更直接的方法。例如，想添加data到Watches窗口，可以用鼠标选中data(选中后显示灰色)，按下鼠标右键，在弹出的快捷菜单中选择Watch 'data'，见右上图则data会自动添加到Watches窗口内。

tsize()成员函数值居然为4，有点不可思议，难道产生的每一个string对象不会超过4个字符的长度？赶紧看看产生的字符窜str的值(把data添加到Watches窗口就可以了，data指向的字符窜就是当前str的值)，是”948izOd5ThYb4zQ”，显然它远远超过4个字符了。接下来的std::memcpy(p, data, sz * tsize());就仅仅复制sz * tsize()个字符到新空间中，此时的sz为1，也就是说仅仅复制1 * 4个字符到新空间中，tsize()的实现很容易找到(选中tsize，按下鼠标右键，从弹出的快捷菜单中选择Find implementation of: 'tsize'，则自动追踪到Satck内成员函数tsize()的实现处)，它是`unsigned tsize() const { return sizeof(T); }`，也就是说sizeof(std::string)的值为4，你敢相信这个结果吗？继续跟踪下去，当企图再往栈中放入第三个std::string对象时，由于空间不足，再次分配更多空间，此时栈数据成员cap的值已经为4, sz已经为2，又进入rebuild函数体，运行语句std::memcpy(p, data, sz * tsize());时程序崩溃了。以上调试过程中如果同时打开栈状态窗口和汇编窗口可能更容易看出问题出在什么地方，当然啦，那需要您能看懂gcc汇编。

或许你已经明白是sizeof惹的祸，其实还不止sizeof, memcpy和memset也功不可没，都有份。如果想改进这个栈的实现就需要替换这两个函数，STL的algorithm函数头下有个copy，是个泛型算法，知道怎么复制简单类型(例如int, char, bool, double等)和复杂类型(例如本例的std::string或者自定义类型)，当然，此刻的sizeof已经不必用了。这里不再介绍怎么改进本例中的栈实现，而是把它留给读者作为练习。

3.5.4 混合风格的程序

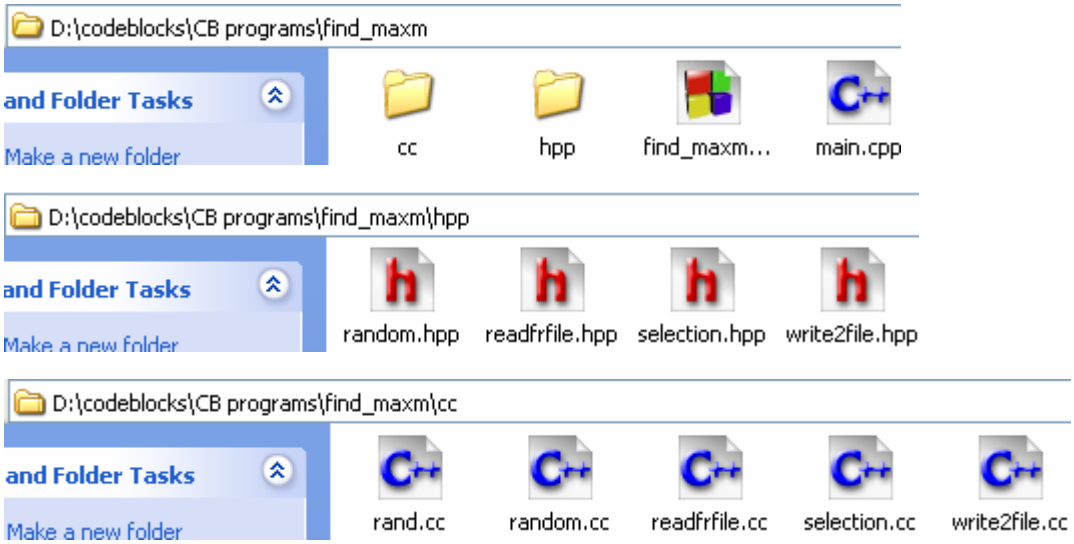
其实很多时候，我们写的程序，尤其复杂的C++程序，并非一种编程风格，而是以上几种编程风格

的混用，而且我们编程经常使用库函数和类。下面给出一个简单例子。

网络上数据非常丰富，有时我们需要从这些海量数据中筛选出我们感兴趣的一些信息。我们在此把这个复杂的问题简单化，假设我们有10G个正整数存放在一个磁盘文件里面，我们需要从这些数中找出1百万个(假设仅仅512MB内存，但是硬盘容量充足)最大的。

为了模拟以上过程，并节约时间，我们处理数据量小一点。首先用随机数产生器产生1千万个正整数，把这些随机数写到硬盘上，放在一个文件里，要求从这些数中找出1K个最大的，在32位系统上使用额外4MB缓存运行速度就基本可以接受。可以这样做：每次读取1M个数放入内存，建造一个堆，筛选出1K个最大的，这样需要读取10次，然后建堆筛选10次，共产生10K个较大的数，再从这10K个中选出1K个最大的。为了加快存取速度，设置读写缓冲区4MB(sizeof(size_t) * 1M)。

这里并不打算再讲述一步步怎么调试程序，因为调试以下程序并不难，但没有给出测试代码，希望读者动手自己补上并调试运行使之得到正确结果。下面是源代码文件层次图和源代码的贴图。



```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <ctime>
#include "hpp/write2file.hpp"
#include "hpp/selection.hpp"

int main()
{
  std::string tofile;
  std::cout << "input a file name to hold random numbers: ";
  std::getline(std::cin, tofile);
  std::ofstream fout(tofile.c_str());
  const size_t SZ = 10000000;     // 产生SZ个随机数
  const size_t M = 1 << 10;       // 找出M个最大的
  const size_t BSZ = 1 << 20;     // 缓存大小为BSZ个元素空间
  std::clock_t time = std::clock();
  write2file(fout, SZ, BSZ);     // 把SZ个随机数写到磁盘上
  std::cout << "generating " << SZ << " numbers and writing them into "
            << tofile << " used: " << std::clock() - time << "ms\n";

  std::clock_t t = std::clock();
  std::ifstream fin(tofile.c_str());
  std::vector<size_t> v;
  v.reserve(SZ / BSZ * (M + 1)); // 设置缓冲区大小
  maxinbuff(fin, M, BSZ, v);      // 找到的最大M个随机数暂存缓冲区
  std::cout << "finding " << M << " maximum numbers in " << tofile
            << " used: " << std::clock() - t << "ms\n";

  std::cout << "input a file name to hold maximum numbers: ";
  std::getline(std::cin, tofile);
  fout.open(tofile.c_str());
  buff2file(fout, v, M);          // 最大的M个随机数写到磁盘上
}
```

```cpp
#ifndef WRITE2FILE_HPP_
#define WRITE2FILE_HPP_

#include <fstream>
#include <vector>
#include "../cc/write2file.cc"

void buff2file(std::ofstream&, const std::vector<size_t>&, size_t);
void write2file(std::ofstream&, size_t, size_t);

#endif


// selection.hpp
#ifndef COLECTION_HPP_
#define COLLECION_HPP_

#include <fstream>
#include <vector>
#include "../cc/selection.cc"

void maxinbuff(std::ifstream&, size_t, size_t, std::vector<size_t>&);

#endif
```

```cpp
// readfrfile.hpp
#ifndef READFRFILE_HPP_
#define READFRFILE_HPP_

#include <fstream>
#include <vector>
#include "../cc/readfrfile.cc"

void readfrfile(std::ifstream&, std::vector<size_t>&);

#endif
```

```cpp
// random.hpp
#ifndef RANDOM_HPP_
#define RANDOM_HPP_

#include <vector>
#include "../cc/random.cc"

// 产生随机数并写入缓冲区中
void gen_rand(std::vector<size_t>&, size_t);

#endif
```

```cpp
// rand.cc
#include <ctime>

class random
{
public:
  explicit random(size_t s = 0) : seed(s)
  {
    if (0 == seed) seed = std::time(0);
    randomize();
  }
  void reset(size_t s)
  {
    seed = s;
    if (0 == seed) seed = std::time(0);
    randomize();
  }
  size_t rand32()//returns a random integer in the range [0, -1UL)
  {
    randomize();
    return seed;
  }
  double real()  //returns a random real number in the range [0.0, 1.0)
  {
    randomize();
    return (double)(seed) / -1UL;
  }
private:
  size_t seed;
  void randomize() { seed = 1103515245UL * seed + 12345UL; }
};
```

```
class rand_help
{
  static random r;
public:
  rand_help() {}
  void operator()(size_t s) { r.reset(s); }
  size_t operator()() { return r.rand32(); }
  double operator()(double) { return r.real(); }
};
random rand_help:: r;

extern void srandom(size_t s) { rand_help()(s); }
extern size_t rand32() { return rand_help()(); }
extern double drand() { return rand_help()(1.0); }
```

```
// random.cc
#include <vector>
#include "rand.cc"

// 产生随机数并写入缓冲区中
void gen_rand(std::vector<size_t>& v, size_t n)
{
  for(size_t i = 0; i < n; ++i)
    v.push_back(rand32());
```

```
// readfrfile.cc
#include <fstream>
#include <string>
#include <cstring>

// 把字符串转化成无符号整数
inline size_t str2uint(const std::string& str)
{
  size_t n = 0;
  for(size_t i = 0; i < str.size(); ++i)
  {
    n *= 10;
    n += (str[i] - '0');
  }
  return n;
}
// 把所有由字符串转化成的整数都写入缓冲区
void str2buff(const std::string& str, std::vector<size_t>& v)
{
  for(size_t i = 0, k; i < str.size(); i = k + 1)
  {
    while(!std::isdigit(str[i])) { ++i; }
    for(k = i; k < str.size() && std::isdigit(str[k]); ++k) {}
    std::string tmp(std::string(&str[i], &str[k]));
    v.push_back(str2uint(tmp));
  }
}
// 每次从文件中读取一行，并把他们转化成整数写入缓冲区
void readfrfile(std::ifstream& fin, std::vector<size_t>& buff)
{
  std::string str;
  std::getline(fin, str);
  str2buff(str, buff);
}
```

```cpp
// selection.cc
#include <vector>
#include <algorithm>
#include <functional>
#include "readfrfile.cc"

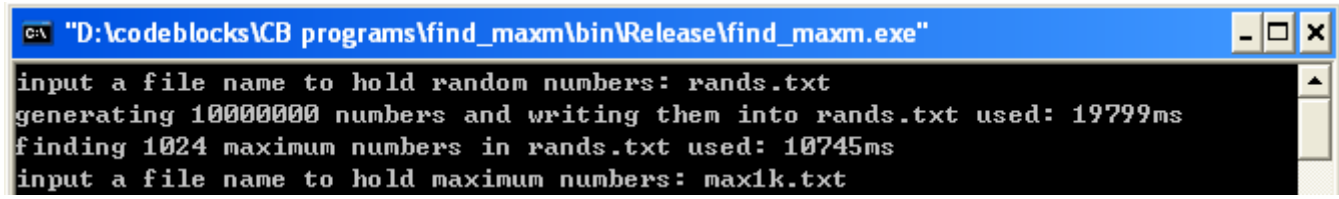
// 使用二叉堆进行筛选，并把筛选结果暂存到缓冲区中
void collect(std::vector<size_t>& v, std::vector<size_t>& b, size_t m)
{
  std::partial_sort(b.begin(), b.begin() + m, b.end(), std::greater<size_t>());
  v.insert(v.end(), b.begin(), b.begin() + m); // 附加到上次存放结果的后面
}

// 每次从文件读取一行，转化成整数，筛出m个最大的候选值暂存起来，
// 待粗筛完毕，然后进行二次筛选，找出最终的m个最大值
void maxinbuff(std::ifstream& fin, size_t m, size_t bsz, std::vector<size_t>& v)
{
  std::vector<size_t> buff;
  buff.reserve(bsz);
  while(!fin.eof())
  {
    readfrfile(fin, buff);
    collect(v, buff, m);
    buff.clear();
  }
  fin.close();
  std::nth_element(v.begin(), v.begin() + m, v.end(), std::greater<size_t>());
}
```

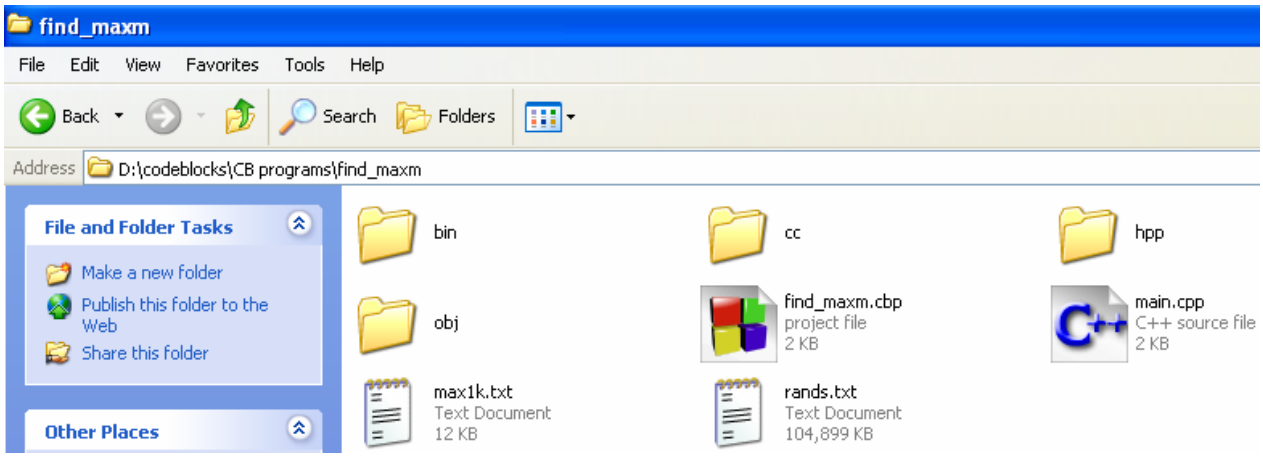
```cpp
// write2file.cc
#include <fstream>
#include <vector>
#include "../hpp/random.hpp"

// 把缓冲区中的前n个数据写入到文件中，然后写入一个换行符
void buff2file(std::ofstream& fout, const std::vector<size_t>& v, size_t n)
{
  if(v.empty())
    return;
  for(size_t i = 0; i < n; ++i)
    fout << v[i] << ' ';
  fout << '\n';
}
// 把n个数据分批写入到一个文件中，每批写入bsz个
void write2file(std::ofstream& fout, size_t n, size_t bsz)
{
  std::vector<size_t> v;
  v.reserve(bsz);
  for(size_t c = 0, T = n / bsz; c < T; ++c)
  {
    gen_rand(v, bsz);
    buff2file(fout, v, bsz);
  }
  const size_t L = n % bsz;
  if(0 != L)
  {
    gen_rand(v, L);
    buff2file(fout, v, L);
  }
  fout.close();
}
```

编译并运行以上程序，一种可能的结果如下截图：



假设生成的随机数存放在文件 rands.txt 中，筛选出的最大 1K 个随机数存放在文件 max1k.txt 中，程序运行完毕，打开所在目录可以看到如下：

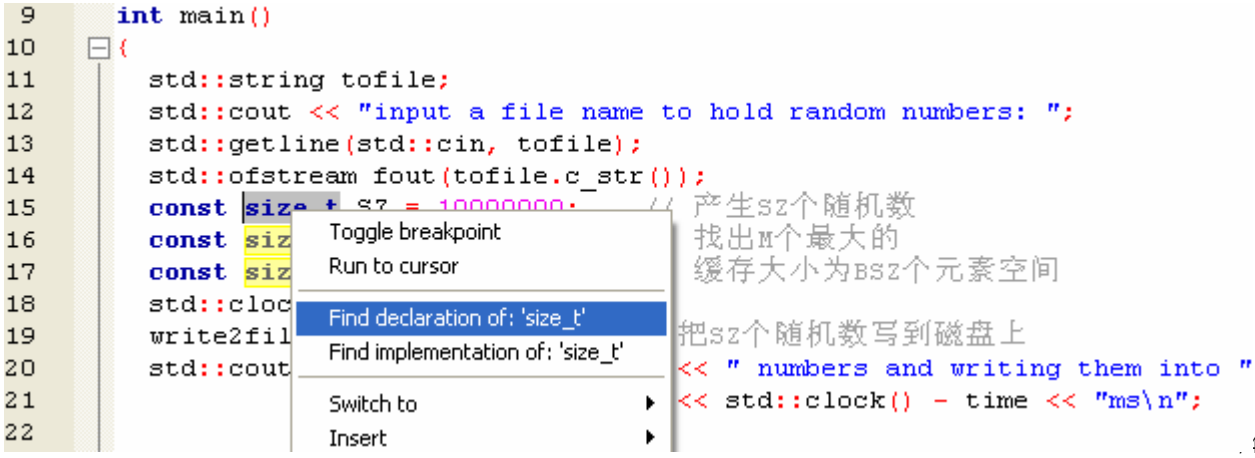


说明可以产生随机数(rands.txt 大小为 104899KB)，也能筛选出一些结果(max1k.txt 文件大小为 12KB)。置于结果是否正确，请读者编程测试验证，作为练习。

以上运行结果是在笔者的古董笔记本电脑(IBM T23)上运行测得，产生一千万个随机数并分批写入磁盘文件消耗了将近 20 秒，分批读取这些数据找出 1K 个最大的大约耗费了 10 秒钟，其实时间主要耗费在读写磁盘文件上，假如不读写磁盘的话，产生 1 千万的随机数在笔者的古董电脑上也不会超过 1 秒，从这些随机数中找出 1K 个最大的需要的时间不会超过 5 毫秒。

3.6 阅读别人编写的程序

有时为了提升自己的水平，我们往往需要阅读别人的程序消化吸收为自己所用。这里仍旧以上面的 3.5.4 小结中的程序作为示例。首先，看几个伪关键字(并非真正的关键字，而是宏定义或者 typedef 定义的别名)。main 函数里面用到了 size_t 和 clock_t，我们看一看它们究竟是什么原型。选中一个 size_t，按下鼠标右键，从弹出的快捷菜单中选择Find declaration of: 'size_t'，见下图。



这样，我们可以找到 size_t 的声明位置，在 stddef.h 头文件里面，是一个 typedef 来的__SIZE_TYPE__，见下面的贴图第 214 行红色方框中的部分。

main.cpp | stddef.h

```
200  #define _BSD_SIZE_T_DEFINED_   /* Darwin */
201  #define _SIZE_T_DECLARED  /* FreeBSD 5 */
202  #define ___int_size_t_h
203  #define _GCC_SIZE_T
204  #define _SIZET_
205  #if defined (__FreeBSD__) && (__FreeBSD__ >= 5)
206  /* __size_t is a typedef on FreeBSD 5!, must not trash it. */
207  #else
208  #define __size_t
209  #endif
210  #ifndef __SIZE_TYPE__
211  #define __SIZE_TYPE__ long unsigned int
212  #endif
213  #if !(defined (__GNUG__) && defined (size_t))
214  typedef __SIZE_TYPE__ size_t;
215  #ifdef __BEOS__
```

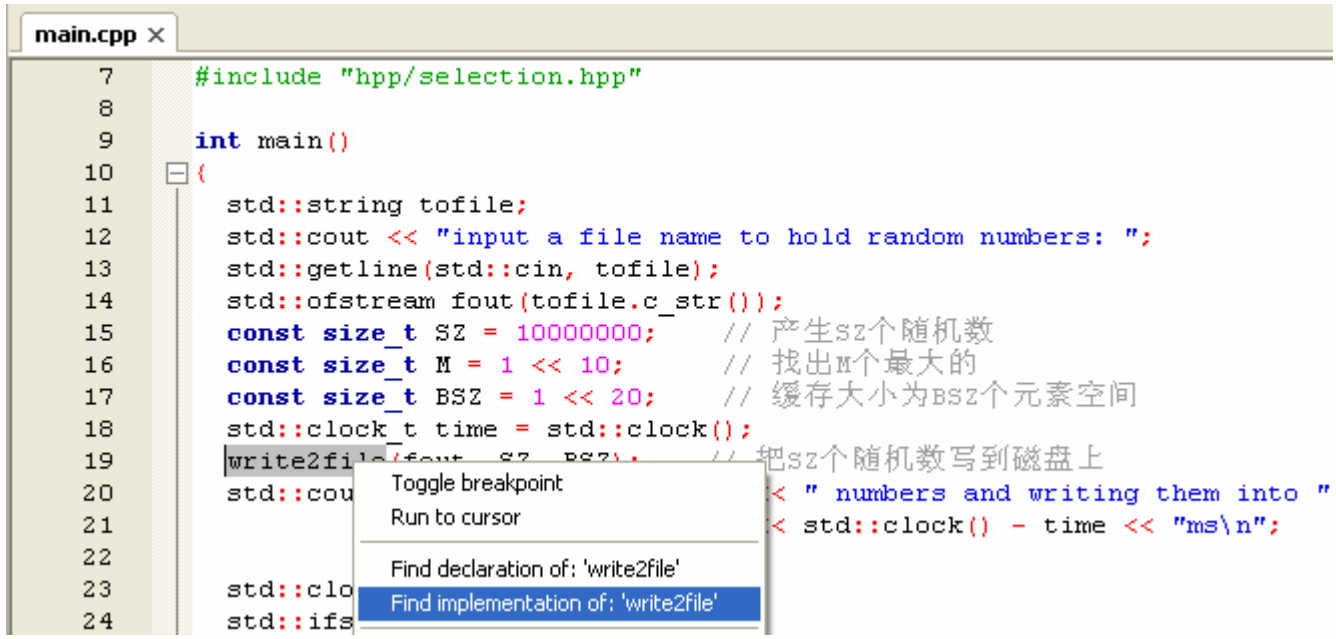
那__SIZE_TYPE__又是什么呢，按照同样的方法追踪下去...，仍然在文件 stddef.h 里面，就在第 211 行，是一个 long unsigned int 类型，用宏定义的，仍然见上图。

那 clock_t 又是什么呢？按照同样的方法去追踪，我们发现它在 time.h 文件里面，是 long 的 typedef，在第 55 行，见下面的贴图红色方框标出的部分。

main.cpp | time.h

```
41  #define _TIME_T_DEFINED
42  #endif
43
44  #ifndef __STRICT_ANSI__
45  /* A 64-bit time_t to get to Y3K */
46  #ifndef _TIME64_T_DEFINED
47  typedef __int64 __time64_t;
48  #define _TIME64_T_DEFINED
49  #endif
50  #endif
51  /*
52   * A type for measuring processor time (in clock ticks).
53   */
54  #ifndef _CLOCK_T_DEFINED
55  typedef long  clock_t;
56  #define _CLOCK_T_DEFINED
57  #endif
```

如果您不明白任何函数名或者定义的变量名均可以采用类似的方法去追踪。假如您想看看某个函数的实现，可以采用类似的方法。例如您想知道 write2file 的实现，就选中这个名称，按下鼠标右键，从弹出的快捷菜单中选择Find implementation of: 'write2file'，见下面的贴图。

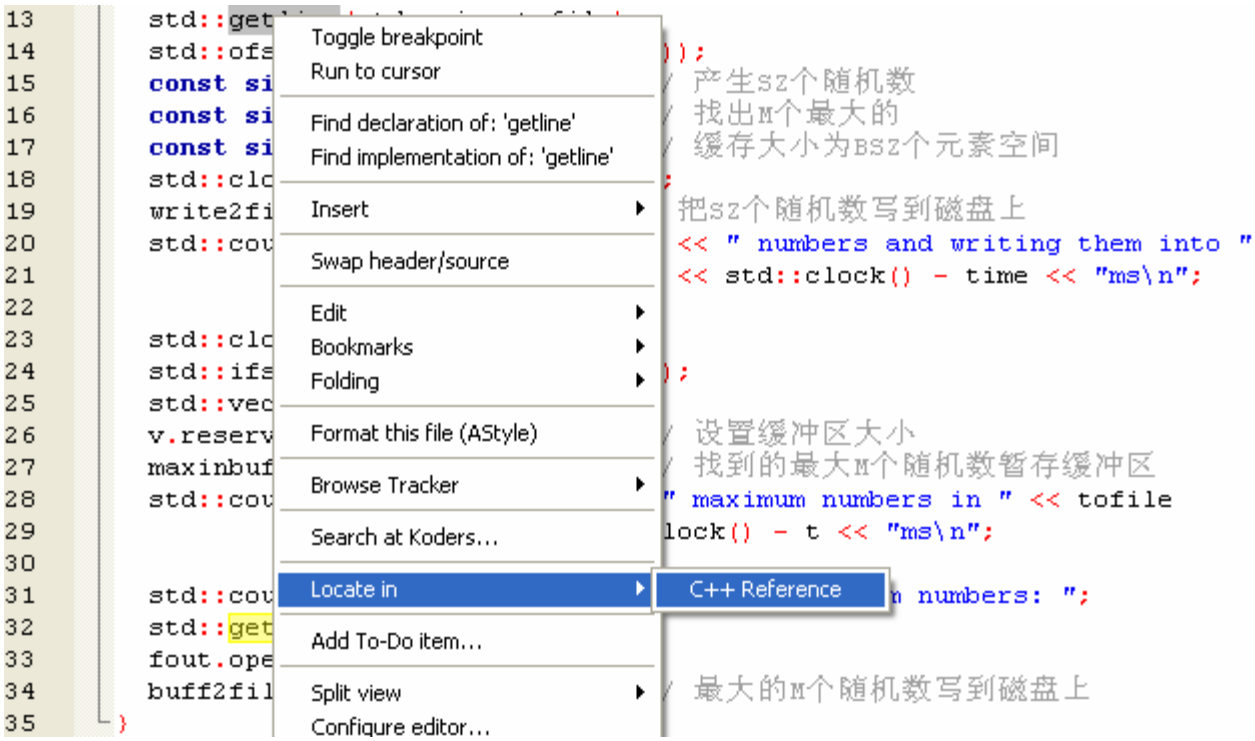


此时会自动找到这个函数的实现，它在文件 write2file.cc 文件里面，光标停留在该函数头前面，见下面的贴图中红色方框标出的部分。如果该实现中用到了其它的函数您不清楚的地方，可以采用此方法继续追踪下去...

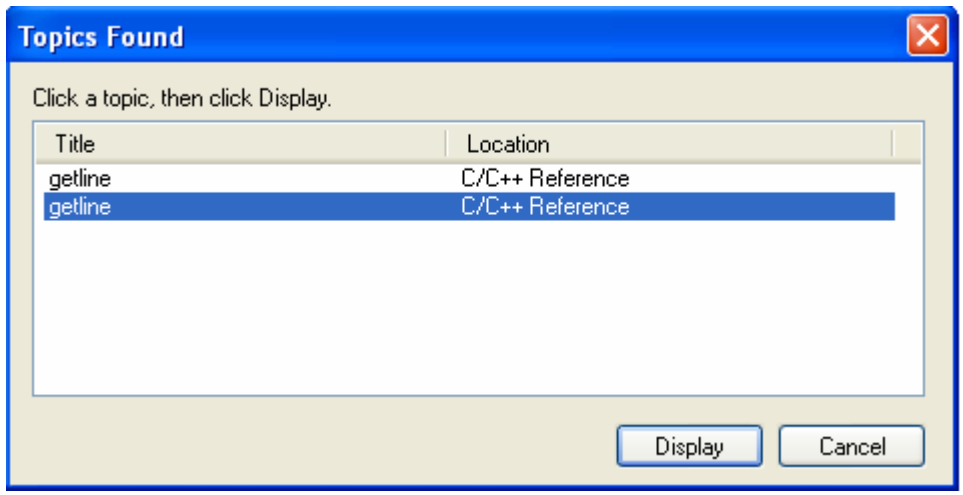
```
// 把n个数据分批写入到一个文件中，每批写入bsz个
void write2file(std::ofstream& fout, size_t n, size_t bsz)
{
  std::vector<size_t> v;
  v.reserve(bsz);
  for(size_t c = 0, T = n / bsz; c < T; ++c)
  {
    gen_rand(v, bsz);
    buff2file(fout, v, bsz);
  }
  const size_t L = n % bsz;
  if(0 != L)
  {
    gen_rand(v, L);
    buff2file(fout, v, L);
  }
  fout.close();
}
```

以上是我们自己定义的函数，如果要是调用的系统的函数怎么办？采用同样的方法固然可以，但是重名函数很多，实现也各不相同，有些函数实现十分复杂，我们很难短时间内看明白。为此我们可以在文档中查询该函数的用法就可以了，可能您不会忘记我们前面有关章节介绍按下 F1 快捷键可以在我们已经事先设置好的 cppreference.chm 里面查询和网上查询。那是一种方法，还有更快捷的方法，我们无需键入函数名就可以自动查找。

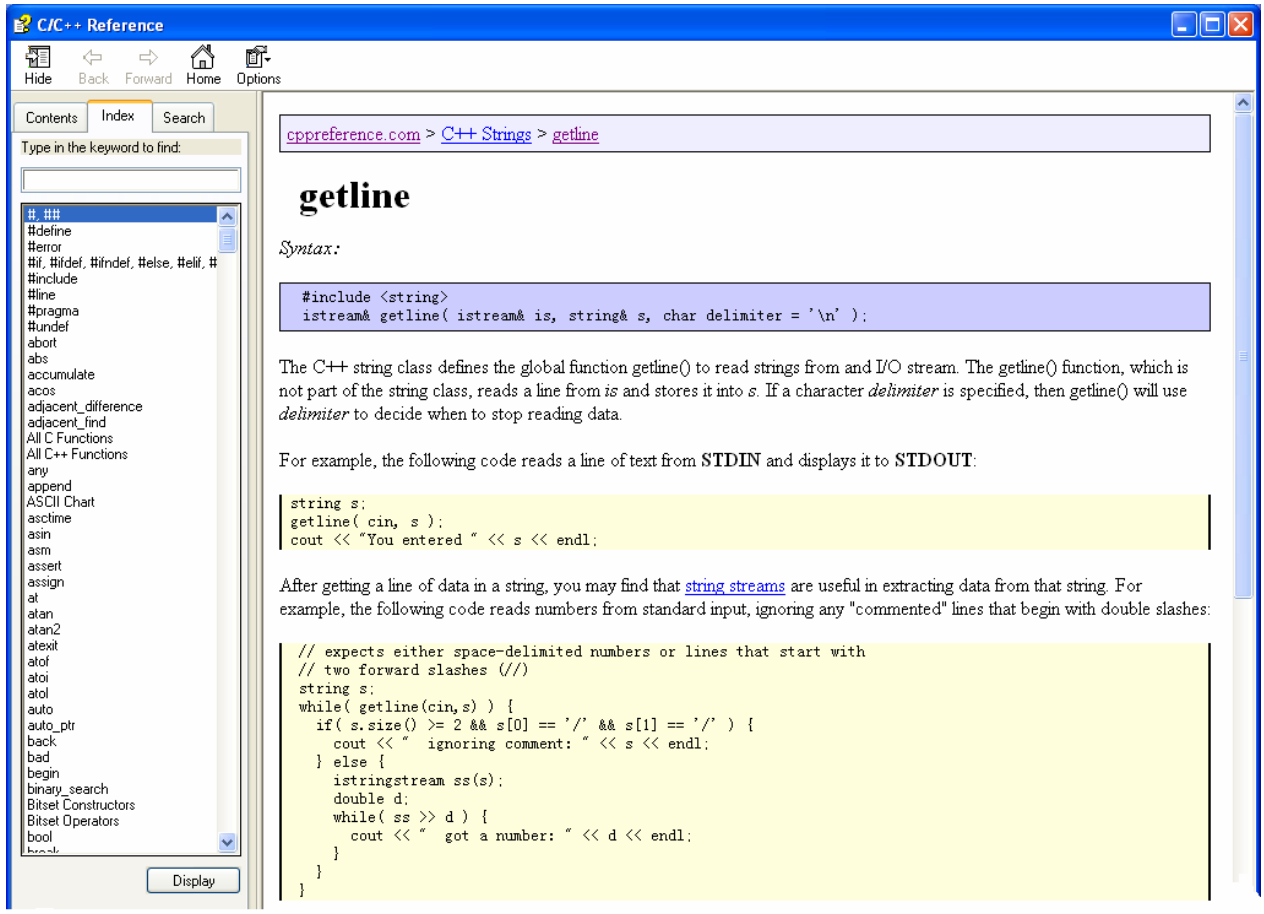
例如，假如我们想查找函数 getline 的相关使用方法，选中 getline 这个名称，按下鼠标右键，在弹出的快捷菜单中选择 Locate in → C++ Reference，这样仍然可以在 cppreference.chm 文件中查找，见下图。



当选择 C++ Reference 的时候，会弹出一个窗口，在帮助文件中找到两个函数名为 getline 的函数，第二个是我们需要的，选中第二个，并点击 Display 按钮，见下面的贴图。



点击 Display 后，会自动搜索 cppreference.chm 文件中有关 getline 函数的相关用法介绍，并切换到该函数的用法界面部分，见下面的贴图。



通过上述介绍的几个简单菜单按钮，我们就可以比较方便的阅读别人编写的源程序了。

4. 附录

附录有几个部分，分别讲述在不同的操作系统上安装 Code::Blocks，配置其他库等。

4.1 在 Linux 下安装 Code::Blocks

Linux 版本众多，不同版本有些差异，这里以在 ubuntu 和 fedora 上安装 Code::Blocks 为例。

4.1.1 Ubuntu

(1) 在/etc/apt/sources.list 下添加如下内容。

以**root**身份登陆，打开一个图形界面的编辑器，在终端上粘贴下面一行内容。

gksu gedit /etc/apt/sources.list

在文件末尾贴上以下内容。

```
# codeblocks
deb http://lgp203.free.fr/ubuntu/ gutsy universe
# wxwidgets
deb http://apt.wxwidgets.org/ gutsy-wx main
```

注意：在第(1)步您也可能需要使用feisty而并非gutsy, 这跟你使用的ubuntu版本有关。

(2) 确保您的package system信任这些源就需要添加他们的key。

在终端键入以下内容。

```
wget –q http://lgp203.free.fr/public.key -O- | sudo apt-key add –
wget –q http://apt.wxwidgets.org/key.asc -O- | sudo apt-key add –
```

在终端键入以下内容更新这些包。

```
sudo apt-get update
sudo apt-get upgrade
```

(3) 安装Code::Blocks

在终端键入下行内容。

```
sudo apt-get install libcodeblocks0 codeblocks libwxsmithlib0 codeblocks-contrib
```

想得到最新的nightly build就重复一次第(3)步。

现在您应该可以在程序语言列表中见到Code::Blocks了。

4.1.2 Fedora

(1) 以**root**身份登陆，在控制台窗口执行下面的命令。

```
yum install codeblocks
```

(2) 运行Code::Blocks

在应用程序的下拉菜单中找到程序设计选择Code::Blocks IDE。

或者打开一个控制台窗口执行下面的命令。

```
codeblocks
```

4.2 在 Mac OS X 下安装 Code::Blocks

4.2.1 准备工作

(1) 确认X11已经安装

看看X11是否在您的系统中已经安装了(在应用程序的菜单里)，如果没有安装就拿出 Mac OS的盘装上吧。安装方法很简单，选中备选安装包的installer，打开X11前面打勾安装上。如果您使用的是10.1至10.3的版本，可以从Apple网上下载X11。

(2) 安装开发工具 (Mac OS X 10.4以上版本不需要这一步)

打开**xterm**., 试试你在终端上不能使用Code::Blocks。xterm是X11的一部分，这也就是为何要安装X11的原因。

运行gcc命令，如果出错信息提示没有输入文件，说明gcc可用，如果出错信息说找不到gcc，那您需要安装开发工具。

如果需要安装开发工具的话，您可以从Apple网站下载或从Mac OS盘上安装，找到Xcode installer装上就是了，运行这个installer，除了文档可以不用安装外，其他全部选中安装上。

Mac OS X 10.3，需要安装1.2以上版本的Xcode工具，Mac OS X 10.4需要安装2.2以上版本的Xcode工具， Mac OS X 10.5需要安装3.1以上版本的Xcode工具。

4.2.2 安装Code::Blocks

解压zip安装包把CodeBlocks.app放到您想放的位置(建议放在/Developer/Applications或者~/Applications下)。

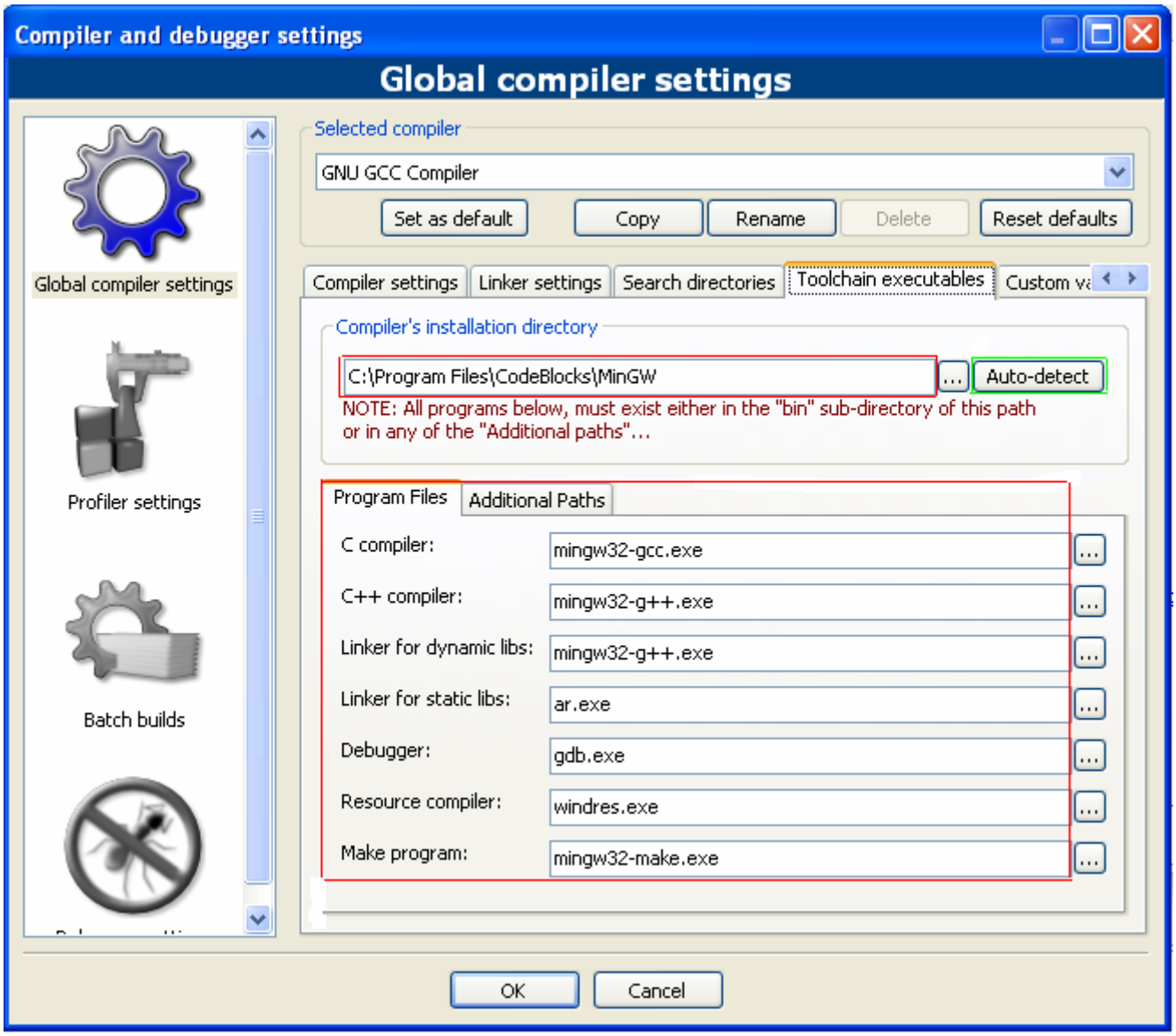
4.3 Code::Blocks 搭配高版本 gcc 编译器

Code::Blocks 搭配较高版本的 gcc 编译器有两种方法，如果你用 Windows 系统，可以升级和配置高版本的 MinGW(内嵌 gcc)从而使用高版本 gcc。在所有系统上都可以升级和搭配高版本的 gcc 编译器。

4.3.1 升级 MinGW

目前最新稳定的 MinGW 版本内嵌了 gcc3.4.5 编译器，测试版则内嵌了 gcc4.3.3，升级 MinGW 前需要先卸载原来古老版本的 MinGW 或者把新版本的 MinGW 安装到不同的位置。

笔者建议您首先卸载古老版本的 MinGW，然后到 http://www.tdragon.net/recentgcc/下载最新测试版 MinGW 的安装器(installer)并执行之，把 MinGW 安装到 Code::Blocks 文件夹下，启动 Code::Blocks，如果不能自动检测到新安装的 MinGW 的话，就在 Settings 下拉菜单中选择 Compiler and debugger…按钮，此时会弹出一个界面，选择左边带有标签 Global compiler settings 的图标，然后选择 Toolchain executables，见下图。用鼠标点击 Auto-detect 按钮(绿色框内)，如果能识别最好，不能识别择需要点击按钮…，手工配置路径(注意红色框内的配置)，配置完毕后，点击 OK 按钮退出就可以了。

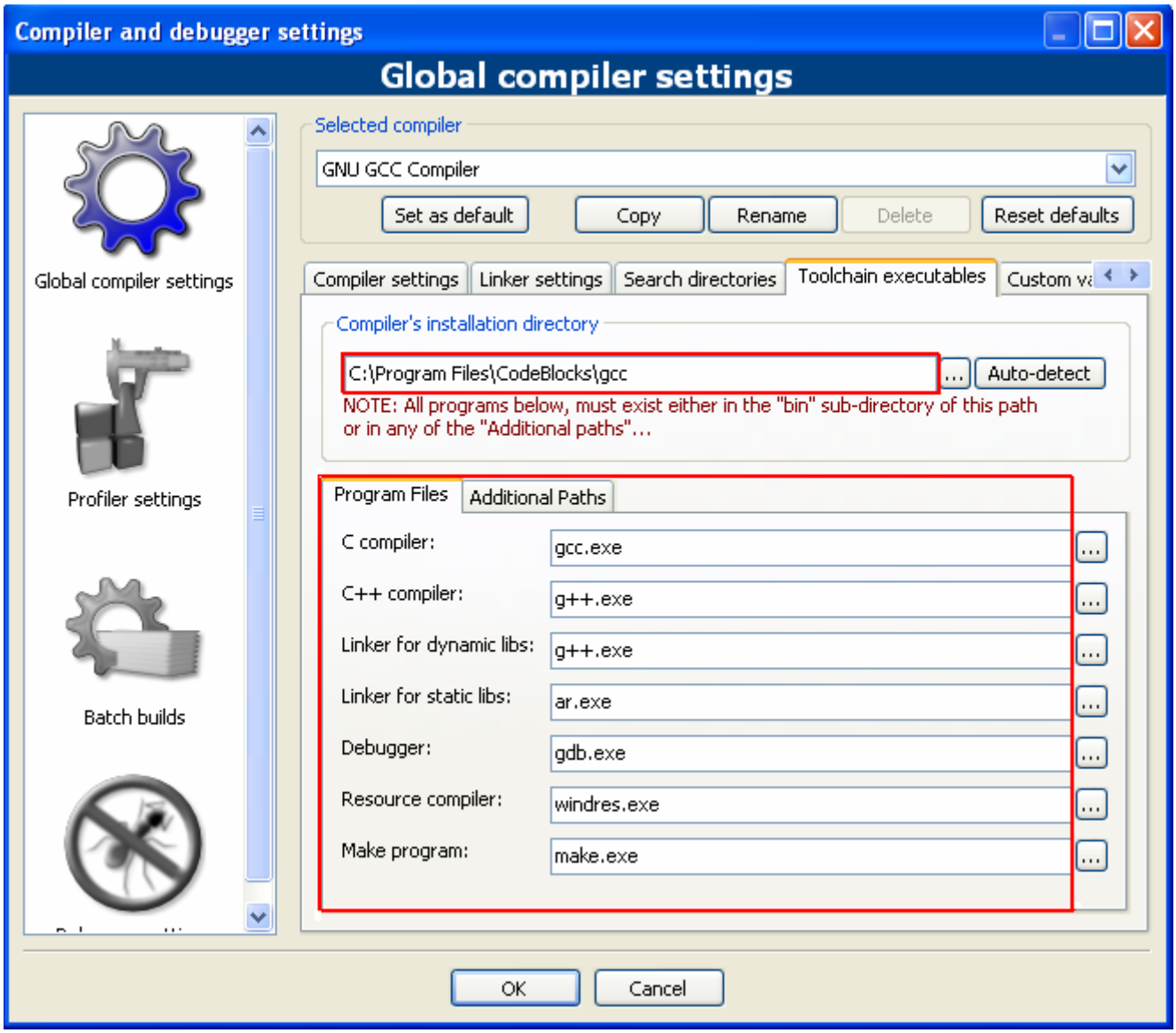


4.3.2 升级并配置 gcc 编译器

如果能找到 gcc 的二进制代码当然最好，如果没有二进制安装包就需要自己编译 gcc，编译 gcc 的方法这里不再赘述，请参阅相关文档。

假设您手头已经有了高版本的 gcc 二进制安装包，Windows 版本的 gcc 二进制文件可以登陆 http://www.equation.com/servlet/equation.cmd?call=fortran 下载。下载完毕后，笔者建议您把它安装到 Code::Blocks 的目录下，然后配置编译器选项(跟上面配置 MinGW 的类似)，从 Settings 下拉菜单中选择 Compiler and debugger…按钮，此时会弹出一个界面，选择左边带有标签 Global compiler settings 的图标，然后选择 Toolchain executables，点击按钮…逐一配置好路径，见下图(注意粗线红色框内的文件配置)。

配置完毕后，编译运行一个小程序测试一下，如果编译成功则说明配置正确，失败就按照上面的配置说明好好检查一下，看看哪里有问题改正。



4.4 安装配置 boost

安装 boost 有点麻烦，因为需要下载源代码和编译文件，编译完 boost，然后再配置，配置完毕后还要测试安装是否正确。以下以 Windows XP 上安装 boost 为例进行介绍。

(1) 下载 boost

登陆 http://www.boost.org 下载最新的 boost 源代码包和编译文件。写本文的时候最新的 boost 源代码包是 1.38.0 版，最新的编译文件 boost jam 3.1.17，下载完毕后解压。

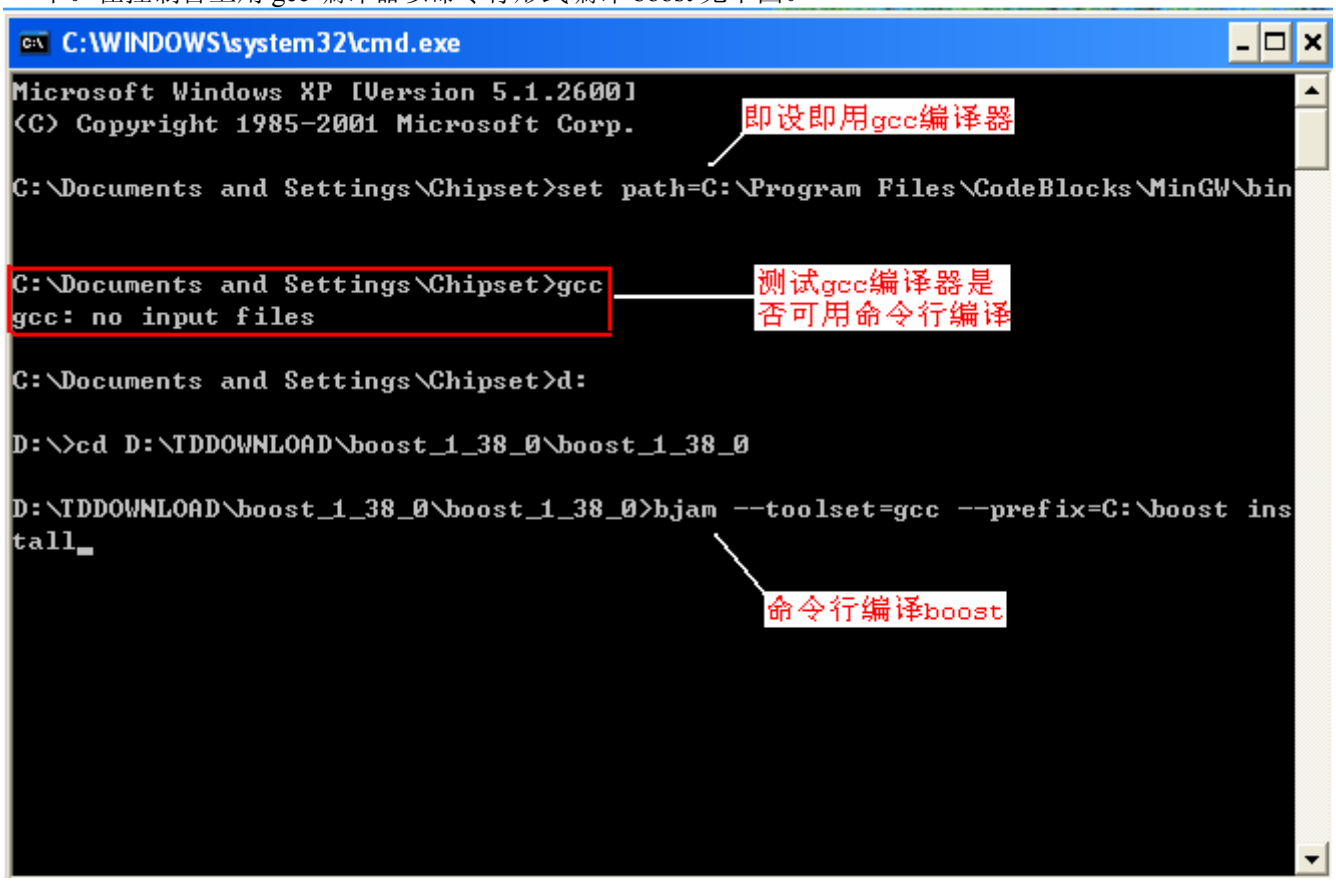
注意最好下载二进制的 boost jam，这样就可以直接使用它编译 boost 了。在 Windows 上安装 boost 需要下载 boost-jam-3.1.17-1-ntx86.zip，在 linux 上安装 boost 就下载 boost-jam-3.1.17-1-linuxx86.tgz，在 Free BSD Unix 上安装 boost 就下载 boost-jam-3.1.17-1-freebsdx86.tgz，Mac OS X 上安装 boost 就下载 boost-jam-3.1.17-1-macosxx86.tgz。

(2) 编译 boost

首先需要解压 boost 1.38.0 的包，再解压 boost-jam-3.1.17-1-ntx86.zip，解压完毕 boost-jam-3.1.17-1-ntx86.zip 后，只有一个可执行文件 bjam.exe，把这个文件复制到解压后的 boost 安装包下(在笔者电脑上是 D:\TDDOWNLOAD\boost_1_38_0\boost_1_38_0)。

接下来需要设计 gcc 编译器环境变量，假如你已经成功安装了 GCC，或者 MinGW，全局环境变量就已经自动配置好了，可以进行命令行编译，否则需要设置 GCC 编译器路径。笔者电脑上 gcc.exe 的路径是 C:\Program Files\CodeBlocks\MinGW\bin 把它加到全局变量路径就可以了。按照这样次序去找，My Computer(我的电脑) -> Properties(属性) -> Advanced(高级) -> Environment Variables(环境变量) -> Path -> Edit(编辑) -> [然后把 gcc.exe 文件的路径(C:\Program Files\CodeBlocks\MinGW\bin)复制到环境变量 Path 其它环境变量的后面]保存就可以了。这是一种一劳永逸的方法，如果您并不经常使用命令行编译文件，可以采用即设即用的方式。打开控制台键入 set path=C:\ProgramFiles\CodeBlocks\MinGW\bin 并按下回车键，则 gcc 编译器本次可用命令行编译，关闭控制台即失效。

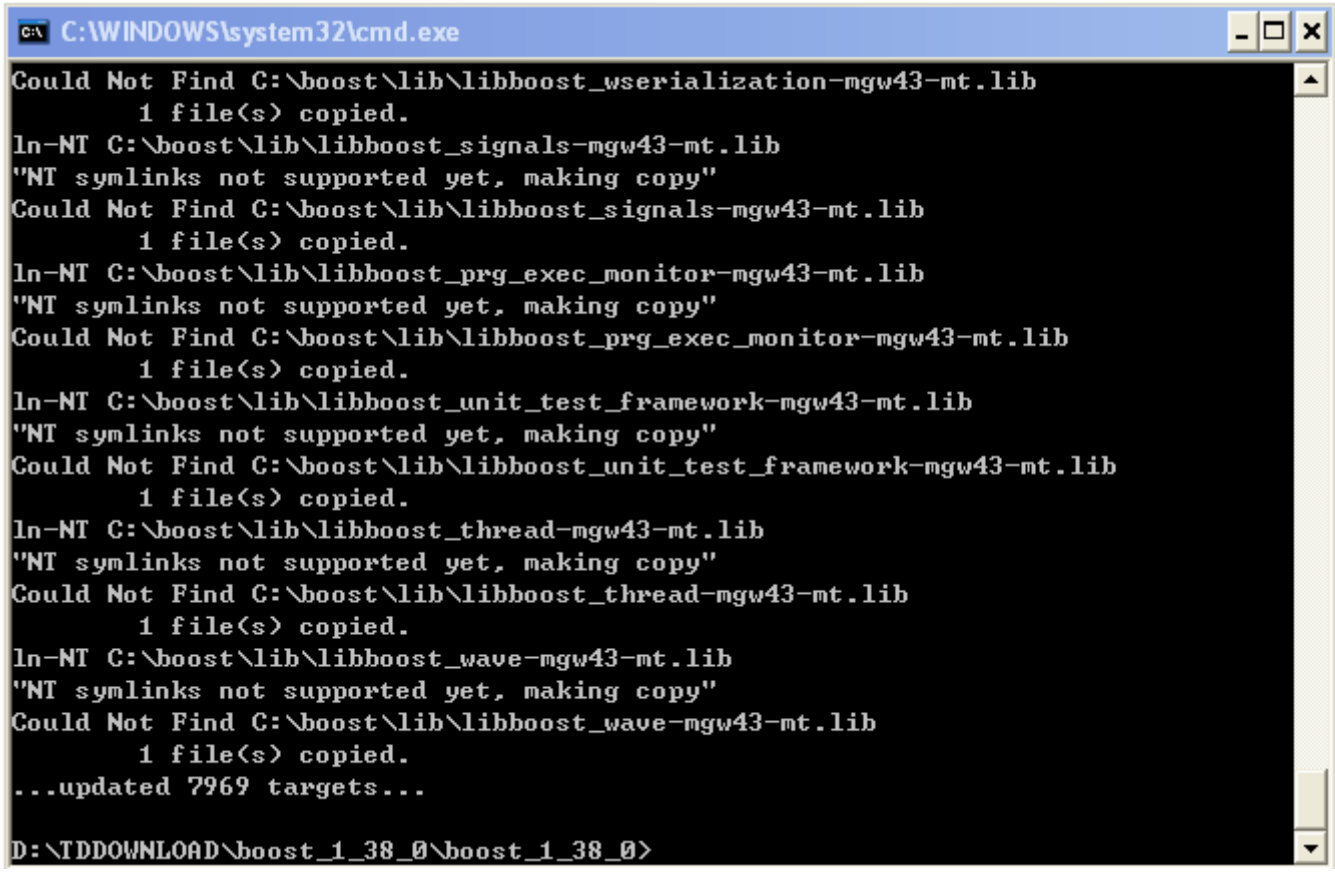
在控制台上找到 bjam.exe 的路径(已经复制到 D:\TDDOWNLOAD\boost_1_38_0\boost_1_38_0 下面)键入 bjam –toolset=gcc –prefix=C:\boost install 进行编译，此命令会把编译后的 boost 文件安装到 C:\boost 下。在控制台上用 gcc 编译器以命令行形式编译 boost 见下图。



需要特别注意：命令行中=前后不可以有空格，文件夹名字随便取，但是一个文件夹名字只能是一个单词，不能是两个或者两个以上中间用空格隔开的单词，否则编译失败。

编译过程将会是比较枯燥等待，好在时间不会太长，编译过程中 CPU、内存、硬盘负荷都很重，在

笔者的古董笔记本电脑(WinXP SP2, PIII 1.13GHz CPU, 512MB 内存 80GB 硬盘)上，纯编译时间大约用了一个小时。编译过程中往往会提示很多信息(包括一些警告信息)，除非失败，否则一律不用理会它们，编译完毕后最终显示结果见下图。



从上图中的文字可以看出，boost1.38.0 成功安装。一般而言，编译成功所有的文件并不容易，根据笔者经验，用 gcc3.4.5 和 gcc4.3.2 编译 boost1.37.0 从未完全编译成功所有的文件，这次使用 gcc4.3.3 编译 boost1.38.0 编译成功的文件数见上图中最后的提示信息(…updated 7969 targets…)。

(3) 配置 boost

打开 code::blocks，在 Settings 的下拉菜单中选择 Compiler and debugger… 然后在 Search directory 栏目下用 Add 添加 boost 的路径(在笔者电脑上是 C:\boost\include\boost-1_38)保存退出。此外，再设置一下全局变量 (在 Setting 下拉菜单下 Global variables… 中的几个值，笔者的设置是 Current Set: boost，Current Variable: boost，base C:\boost\include\boost-1_38)。

(4) 测试 boost

我们仅仅需要测试 boost 是否安装配置正确，而并非测试 boost 是否完整或者有没有 bug，测试 boost 的完整性和是否有 bug 不是一件很容易的事情，boost 团队中有很多人一直进行着这项伟大的工作。

```cpp
//简单测试，仅仅测试是否安装正确，而不是测试boost完整性
//以下程序的功能是从终端接受一个整数序列，然后乘以3再输出到终端
//test1.cpp
#include <iostream>
#include <iterator>
#include <algorithm>
#include <boost/lambda/lambda.hpp>

int main()
{
  using namespace boost::lambda;
  typedef std::istream_iterator<int> in;
  std::for_each(in(std::cin), in(), std::cout << (_1 * 3) << " ");
}

```

```cpp
//以下程序在终端输出[2](2,8)
//test2.cpp
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/io.hpp>
#include <iostream>

using namespace boost::numeric::ublas;

int main ()
{
  vector<double> x (2);
  x(0) = 1; x(1) = 2;
  matrix<double> A(2, 2);
  A(0, 0) = 0; A(0, 1) = 1;
  A(1, 0) = 2; A(1, 1) = 3;
  vector<double> y = prod(A, x);
  std::cout << y << std::endl;
  return 0;
}

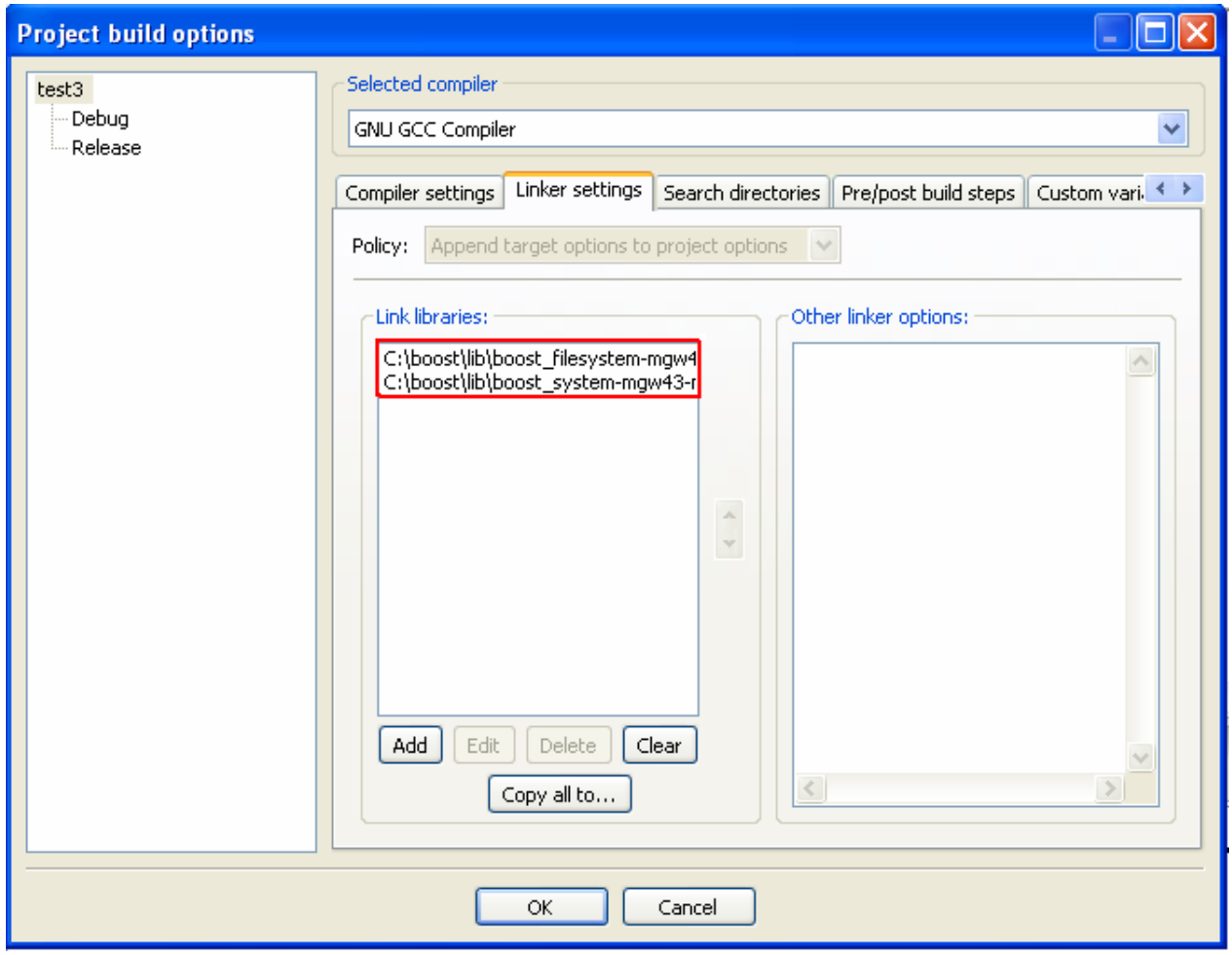
```

```cpp
/*******************************************
  test3.cpp
  注意编译运行时需要连接两个库文件
  boost_system-mgw43-mt-1_38.dll和
  boost_filesystem-mgw43-mt-1_38.dll
*******************************************/
#include <iostream>
#include <boost/filesystem/operations.hpp>

namespace bfs = boost::filesystem;
int main()
{
  bfs::path p("test3.cpp");
  if(bfs::exists(p))
  std::cout << p.leaf() << std::endl;
}

```

编译和运行 test3.cpp 都需要连接两个库文件，编译前需要在本工程中添加上这个两个文件的路径，见下图红色框框起来的两个文件(首先在 Project 下拉菜单中选择 Build options…，然后可以参考下图用 Add 按钮添加，见红色框内文字)。



编译成功后，运行时还需要能找到这两个文件。

4.5 安装配置 Qt

C/C++ GUI 库多如牛毛，商用的免费的都很多。其中名气最大的 C/C++ GUI 库可能就是 Qt 了， Qt 非常强大而且版本升级很快，既有商用版也有免费版。

(1) 下载

如果希望在 Code::Blocks 下开发 Qt 的应用，首先需要到 Qt 的官方网站根据您使用的系统下载相应 Qt 的最新版本 http://www.qtsoftware.com/downloads，然后进行安装。如果您希望在 Windows 下用 Qt 编写应用程序，一般可以选用两种版本 MinGW 的或者 MSVC 的，这里以 MinGW 的为例进行简单介绍安装配置。

(2) 安装

下载完毕，启动安装文件进行安装，安装过程中可选是否安装 MinGW，因为我们已经安装了 MinGW 因此不需该项(去掉前面的勾)，笔者建议您把 Qt 安装在 CodeBlocks 目录下。Qt 的库相对比较大，安装时间可能比较长，请耐心等待。

(3) 设置系统环境变量

安装完 Qt，为了便于搜索需要加载的文件，需要设置系统环境变量，操作系统的环境变量设置要特别小心，否则很难说会出现什么问题。以笔者电脑安装的 Qt4.5 为例，拖动鼠标到我的电脑图标上，按下右键弹出菜单选择属性，再从弹出菜单中选择高级，然后选择环境变量，填入以下这些项目。

Qtdir=C:\Program Files\CodeBlocks\Qt\qt

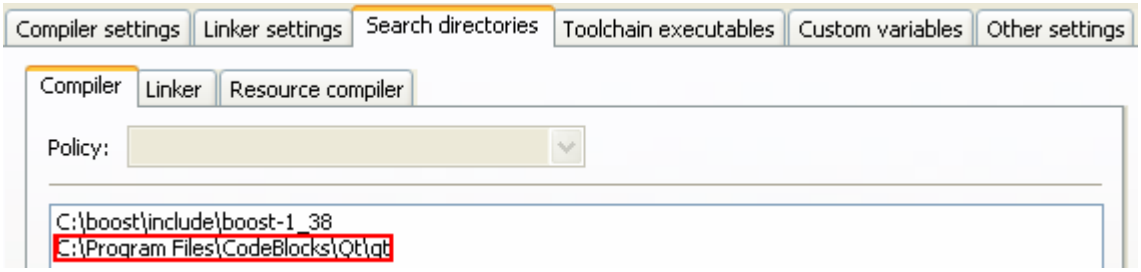
Include=C:\Program Files\CodeBlocks\Qt\qt\include;C:\Program Files\CodeBlocks\MinGW\include;

Lib=C:\Program Files\CodeBlocks\Qt\qt\lib;C:\Program Files\CodeBlocks\MinGW\lib;

Path 中加入.;C:\Program Files\CodeBlocks\Qt\qt\bin;C:\Program Files\CodeBlocks\MinGW\bin;

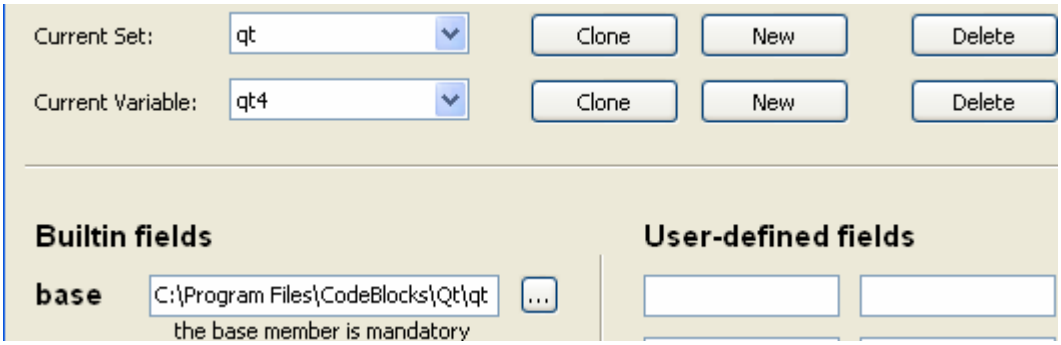
(4) 设置编译器搜索路径

Setting 下拉菜单下选择Compiler and debugger...然后在 Search directory 栏目下用 Add 添加 Qt 路径(笔者电脑上 C:\Program Files\CodeBlocks\Qt\qt)，见下面截图红色框内文字。



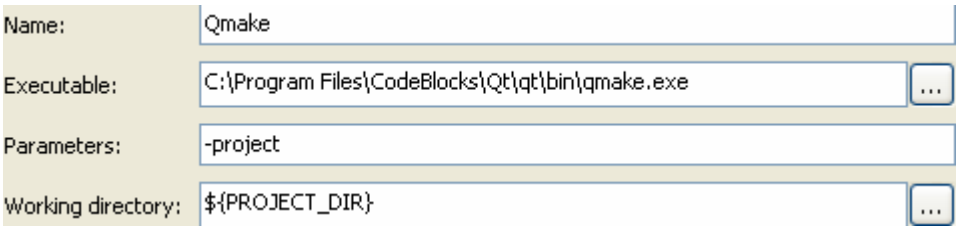
(5) 设置 Code::Blocks 全局变量

Setting 下拉菜单下选择Global variables...在弹出的对话框中设置 Qt 的全局环境变量。其中有一项 base 栏目必填，该项是 Qt 的安装路径(笔者电脑上是base C:\Program Files\CodeBlocks\Qt\qt)，见下图。



(6) 添加 Qmake

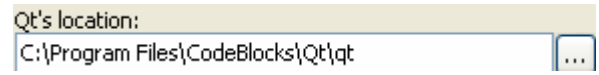
打开 Code::Blocks 的 Tools 下拉菜单，选择Configure tools...添加一个 Qmake，填入选项见下面贴图。



再添加一个 Qmake，但是不用写参数(Parameters:后面的框不填-project，空着)。

(7) 测试

使用 Code::Blocks 的 Wizard 创建一个 Qt 工程。工程名任意，接下来有一个选项需要填入 Qt 路径，这个需要特别注意，否则编译器搜索不到需要加载的文件(在笔者电脑上的路径是 C:\Program Files\CodeBlocks\Qt\qt)见右面的截图。

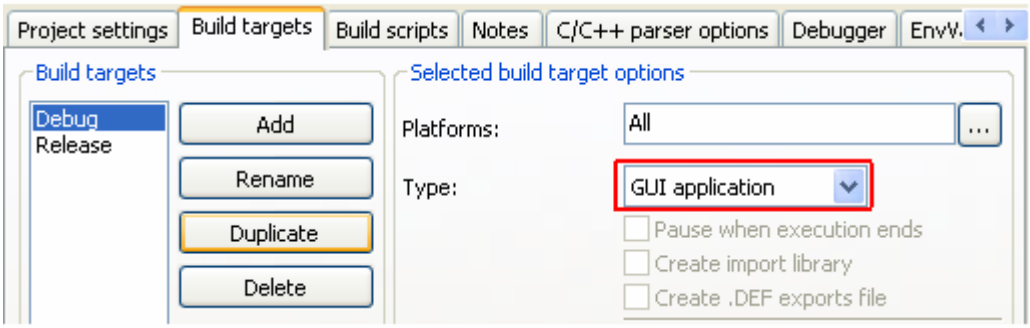


编译器选择 MinGW，最后产生的代码见下面贴图。

main.cpp

```
1  #include <QApplication>
2  #include <QFont>
3  #include <QPushButton>
4
5  int main(int argc, char* argv[])
6  {
7    QApplication app(argc, argv);
8
9    QPushButton quit("Quit");
10
11   quit.resize(75, 30);
12   quit.setFont(QFont("Times", 18, QFont::Bold));
13
14   QObject::connect(&quit, SIGNAL(clicked()), &app, SLOT(quit()));
15
16   quit.show();
17
18   return app.exec();
19 }
20
```

配置好该工程的编译选项(Project 下拉菜单选择 Build options…设置适当编译选项)和属性(注意 Project 下拉菜单 Properties…下 Build targets 类型选择 GUI application，见下面截图红色框内文字)。



编译之，除了几个警告外一般不会出现什么问题。假如您以上各个环节配置正确，运行编译后的二进制文件结果见下图(左)。



特别说明

笔者编译和调试本书中所有的程序都使用测试版 MinGW(内嵌了 gcc4.3.3)，如果您想编译本书中的任何程序，笔者也建议您使用较新版本的编译器(例如 g++4.3.3 或者 g++4.4)和调试器(例如 gdb6.8.3)，此外，编辑部分程序使用 Code::Blocks build 5456 (2009 年 02 月 14 日的 nightly build 版)，编辑部分程序使用 Code::blocks8.02 官方版。

作者对本书中的任何程序代码不提供任何形式的担保，如果您希望在与本书无关的地方使用部分或者全部代码，为了不致于给您、我以及他人带来不便，代码前请务必加上以下文字信息。

```
/**************************************************************************
  Copyright (C) 2009 Chipset

  This program is free software: you can redistribute it and/or modify
  it under the terms of the GNU Affero General Public License as
  published by the Free Software Foundation, either version 3 of the
  License, or (at your option) any later version.

  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
  GNU Affero General Public License for more details.

  You should have received a copy of the GNU Affero General Public License
  along with this program. If not, see <http://www.gnu.org/licenses/>.
**************************************************************************/
```

如果您阅读本书过程中发现错误不妥之处请给我写信 Email: chipset0418@yahoo.com.cn 或者登陆我的博客 http://www.cppblog.com/chipset 给我留言。如果您对本书有什么建议，Chipset 洗耳恭听。

如果您在使用 Code::Blocks 的过程中发现本书能给您带来一点心得或收获，那将是本书作者 Chipset 最大的荣幸。

谢谢！